PIXELES

Forja tu propio motor gráfico con OpenGL y C++

Oscar J. Espinosa 2025

CONTENIDOS

1. INTRODUCCIÓN 8

1.1. ¿QUÉ ES UN MOTOR GRÁFICO? 8

1.2. LO QUE ENCONTRARÁS AQUÍ 8

1.3. Antes de empezar 9

1.4. Formato 9

2. UNA VENTANA A OPENGL 11

2.1. ¿Qué es OpenGL? 11

2.2. GLFW 12

2.2.1. GLFW en Windows 12

2.2.2. GLFW en Linux 13

2.3. GLAD 14

2.4. GLM 15

2.5. Hora de programar 15

2.6. Un poco de color 18

2.7. Tomando control 19

3. UNA ESCENA CUBISTA 21

3.1. Vértices y triángulos 21

3.2. Aquel que pinta: El shader 23

3.2.1. Vertex Shader 23

3.2.2. Fragment Shader 24

3.2.3. Geometry Shader 24

3.3. Transformando el mundo 24

3.3.1. Matriz Modelo 25

3.3.1.1. Traslación 25

3.3.1.2. Rotación 26

3.3.1.3. Escalado 27

3.3.1.4. Combinación de matrices 27

3.3.2. Matriz Vista 28

3.3.3. Matriz Proyección 28

3.3.3.1. Ortogonal 29

3.3.3.2. Perspectiva 29
3.4. Buffers, el almacén 30
3.4.1. VBO: Vertex Buffer Object 30
3.4.2. EBO: Element Buffer Object 31
3.4.3. VAO: Vertex Array Object 31
3.5. A pintar cubos 32
3.5.1. Creando los shaders 33
3.5.1. Creando las cajas 39
3.5.2. Creando la escena 41
3.5.3. El Z-Buffer 46
4. HORA DE ORGANIZAR 47
4.1. El Resource Manager, tu secreatario 47
4.1.1. Recurso Genérico 48
4.1.2. Recurso Shader 49
4.1.3. Recurso Malla 50
4.1.4. Recurso Modelo 51
4.1.5. Punteros Inteligentes 54
4.1.6. El Gestor de Recursos 54
4.2. El Scene Tree, tu director 57
4.2.1. Nodos del árbol 58
4.2.2. Entidad Genérica 63
4.2.3. Entidad Modelo 63
4.2.4. Entidad Cámara 66
4.2.4.1. Rotación de Euler 67
4.2.4.2. Vector Front usando Euler 68
4.2.4.3. Mirando por la cámara 70
4.2.4.4. Moviendo la cámara 71
4.2.4.5. Construyendo la cámara 74
4.3. Adaptando nuestros recursos 74
4.4. Una escena cubista y ordenada 75
4.4.1. El Delta Time 79
4.4.2. El bucle de renderizado ordenado 81

4.4.3. Moviéndonos por la escena 82
5. ILUMINANDO LA ESCENA 85
5.1. La luz de Phong 85
5.1.1. Luz Ambiental 85
5.1.2. Luz Difusa 86
5.1.3. Luz Especular 87
5.1.4. Combinado de Phong 89
5.2. Calculando normales y extrañas 89
5.3. Preparando el Vertex Shader 94
5.4. El sol, la bombilla y la linterna 95
5.4.1. Luz Direccional 96
5.4.2. Luz Puntual 96
5.4.3. Luz Focal 97
5.4.4. Entidad Luz 98
5.5. Complicando el Fragment Shader 100
5.6. Dibujando las luces 105
5.7. Texturizando el cubo 108
5.7.1. Wrapping 111
5.7.2. Filtering 112
5.7.3. Mipmaps 112
5.7.4. Dibujando las texturas 113
5.8. Una escena cubista, ordenada e iluminada 116
6. COMPLICANDO LAS MALLAS 123
6.1. Modelos con Assimp 123
6.1.1. Assimp en Windows 123
6.1.2. Assimp en Linux 124
6.1.3. Leyendo la escena con Assimp 124
6.2. Modelos en nuestra escena 130
6.3. Face Culling 132
6.3.1. Cambiando el sentido del cubo 134
6.4. Siéntete observado 136
6.5. Dibujando el cielo: El Skybox 140

6.5.1. Shaders para el Skybox **142**
6.5.2. El cielo de la escena **143**
7. FORJANDO EL MOTOR 149
7.1. Puntos clave de una librería **149**
7.2. Escondiendo la complejidad **151**
7.3. Compilando con CMAKE **161**
7.4. Un usuario entra en la escena cubista **164**
8. DIARIO DE VIAJE 168
8.1. Mirando hacia el futuro **168**
8.1.1. Las sombras **169**
8.1.2. Transparencias **169**
8.1.3. Animaciones **169**
8.1.4. Carga de niveles **169**
8.1.5. Un pequeño motor llamado Inazuma **170**
8.2. Bibliografía **170**
8.2.1. Bibliografía del libro **170**
8.2.2. Bibliografía recomendada **171**
8.3. Glosario **171**

1. INTRODUCCIÓN

¡Buenas! Si estás aquí es probable que estés interesado en el mundo de la renderización de gráficos 2 y 3D a tiempo real (¿y quién no?). Pronto te habrás dado cuenta de que programas como Unreal Engine hacen poco por explicarte cómo está ocurriendo esa magia, arrastras blueprints y llamas a funciones, ¿pero qué engranajes se estarán moviendo por detrás para renderizar ese cubo? Si quieres averiguarlo y crees que la mejor forma es haciéndolo tú mismo, de forma práctica, este es tu libro.

1.1. ¿QUÉ ES UN MOTOR GRÁFICO?

Antes de nada, definamos así rápido qué es un motor gráfico, que igual leyendo "Unreal Engine" te has asustado pensando que en este libro vamos a hacer el trabajo que una empresa multimillonaria ha tardado años en desarrollar. Aunque Unreal Engine, Unity o Godot son efectivamente motores gráficos, SFML, Raylib o SDL aunque sean "sólo librerías" también lo son.

¿Entonces, si pinta cosas por pantalla ya es un motor gráfico? Em... bueno, más o menos. Ambos tipos de motor cumplen con ser un conjunto de herramientas y librerías que facilitan la creación de aplicaciones gráficas. La verdadera diferencia está en el nivel de abstracción.

Mientras que motores como Unreal Engine son un entorno de desarrollo completo, puedes crear una aplicación entera de principio a fin dentro del motor, librerías como SFML son mucho más humildes. Te dan las herramientas para que tú puedas dibujar formas básicas, cargar imágenes o modelos, mover la cámara, etc. Tú eres el director de la escena, ellos te proporcionan las herramientas para construirla. En este libro vamos a invertir ese orden: nosotros crearemos las herramientas para que nuestros usuarios las usen.

1.2. LO QUE ENCONTRARÁS AQUÍ

Este libro te guiará a través de un viaje en el que comenzarás con un archivo vacío y terminarás con una librería que podrás importar en cualquier otro proyecto. Con ella, podrás renderizar formas geométricas básicas, imágenes 2D

e incluso modelos 3D. Controlarás la iluminación, la cámara y la escena completa. ¡Todo el poder en tus manos!

¿Suena muy dificil? Correcto, lo es, pero te acompañaré durante todo momento explicando cada uno de los detalles importantes a tener en cuenta. Dicho esto, has de tener en cuenta que este no es un libro de OpenGL, C++ ni de matemáticas, muchas veces simplificaré conceptos porque no son relevantes para nuestro objetivo.

No te preocupes, también te indicaré cuándo estoy resumiendo algún tema para que si te pica la curiosidad investigues por tu cuenta y dejaré al final de cada capítulo el código fuente para que puedas revisarlo siempre que quieras.

1.3. ANTES DE EMPEZAR

Nuestro lenguaje de programación será C++, por lo que tanto su instalación como una comprensión básica del mismo serán necesarios. No necesitas tampoco saber cada entresijo (buena suerte intentándolo) del lenguaje, simplemente saber la sintaxis básica y las bases de la programación orientada a objetos. Si prefieres profundizar un poco más, te recomiendo *learncpp*[1] en el caso de que sepas inglés. Si no, hay numerosas fuentes en español[2].

No será necesario que conozcas OpenGL ni que sepas de las matemáticas implicadas en la renderización de gráficos. Como comenté antes, en este libro estos temas se tratarán de manera superficial, pero si quieres profundizar te recomiendo *learnopengl*[3] en inglés y *opengl-tutorial*[4]*en español*.

1.4. FORMATO

Durante el trascurso del libro crearás, de forma incremental y práctica, tu motor gráfico poco a poco. Cada uno de los capítulos se centrará en un aspecto en específico, puedes tomarte cada una como una práctica individual, aunque por su naturaleza secuencial será dificil saltárselos si un apartado no te interesa.

---

[1] Learncpp
[2] Por ejemplo, el cortito vídeo de Sinergia | CURSOS
[3] Learnopengl
[4] OpenGL Tutorials

No te preocupes si algo se te hace cuesta arriba, como te he comentado al final de cada capítulo dejaré un link al repositorio con el código fuente del producto final que deberías alcanzar en cada capítulo para que puedas revisarlo y jugar con él.

Además del texto explicativo, añadiré comentarios con detalles o consejos opcionales por si quieres investigar más por tu cuenta

```
// Así se verá si pongo
// un ejemplo de código
int main(){
    return 0;
}
```

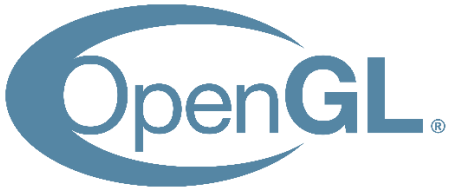
Bueno, todo listo y explicado, ¿no? Genial, empecemos.

2. UNA VENTANA A OPENGL

Ya he mencionado un par de veces a nuestro amigo OpenGL, y eso es porque imagino que si estás aquí no será un término desconocido para ti. Aun así, vamos a repasar qué es exactamente OpenGL, su utilidad, y todo lo necesario para hacer nuestra primera ventana. Intentaré no ser demasiado denso y teórico.

2.1. ¿QUÉ ES OPENGL?

OpenGL es una API (Interfaz de Programación de Aplicaciones) que te provee con una amplia gama de funciones que te permiten interactuar con el Hardware gráfico.



En realidad, más que la API en sí OpenGL es la especificación de dichos métodos. Esta especificación contiene todos los inputs y outputs de cada función, su funcionamiento y su rendimiento. De esta forma, los fabricantes de las tarjetas gráficas (o cualquier desarrollador lo suficientemente valiente) son los encargados de desarrollar la API siguiendo dicha especificación. Esta es una de las razones por las que deberías tener los drivers de tu gráfica siempre actualizados

Lo más importante a tener en cuenta de OpenGL es que es una máquina de estados. Es decir, OpenGL tiene un contexto que contiene cómo debe comportarse en cada momento, su estado actual. Modificando ese contexto se modificará su comportamiento. Si le dices al contexto que los pixeles sean amarillos, cada vez que se dibuje un pixel lo hará de amarillo.

Este detalle es importantísimo de entender, trabajar con OpenGL implica cambiar de estado continuamente y ser consciente siempre del contexto actual, de lo contrario ocurrirán bugs visuales y cosas fuera de nuestro control.

Otro detalle a recalcar es que las librerías comúnmente están escritas en C, aunque luego tenga derivaciones a un lenguaje de más alto nivel. Cuando trabajemos directamente con la librería o creemos nuevos objetos, tendremos que respetar su sintaxis.

OpenGL usa tipos como GLfloat y GLint que nacen de estos tipos básicos de C para garantizar tamaños fijos, de esta forma se puede mantener las compatibilidades en diferentes plataformas.

Más adelante hablaré más sobre conceptos internos de OpenGL, de momento con que sepas el contexto básico es suficiente. Si quieres una información más profunda repasa las recomendaciones del apartado 1.2.

2.2. GLFW

Genial, pues si OpenGL es sólo una especificación, ¿cómo vamos a crear nuestra ventana? Bueno, lo primero que necesitamos es decirle a nuestro sistema operativo que queremos crear una ventana para dibujar y definir un contexto para OpenGL.

Como te imaginarás, al ser dependiente de nuestro sistema operativo, OpenGL no se moja las manos al especificar estos métodos. Para ello existen diversas librerías que, orientadas a OpenGL, nos permiten estas funcionalidades junto a otras también útiles como el control de inputs del usuario. Aquí usaremos GLFW, aunque siéntete libre de usar cualquier otra librería (GLUT, SFML...) e ir adaptando el código.

2.2.1. GLFW en Windows

Vamos a suponer que usas el IDE de Visual Studio (2022) y si no, será fácil de adaptar los pasos a otro entorno. Lo primero será bajar los binarios pre-compilados de Windows en la página de GLFW[5]. En estos archivos veremos que se encuentran varias carpetas, localiza la que tenga el nombre de tu IDE y su versión, esa y la carpeta "include" son las importantes.

En tu proyecto, creas una carpeta llamada "Librerias" (o algo así, es sólo una recomendación) y añade dentro otra para GLFW. Después, creas otras dos

[5] Página de GLFW

dentro: “include” y “lib”. Dentro de “include” meterás todo el contenido de la carpeta del mismo nombre que encontraste en los archivos que te descargaste antes. Por otro lado, en “lib” deberán ir todo el contenido de la carpeta que tenía la versión de tu IDE.

Una vez hecho esto, desde el explorador de soluciones haces click derecho en tu proyecto y vas a propiedades. Asegúrate de añadir arriba para todas las configuraciones y para todas las plataformas. Después en C/C++/General encontrarás la parte de “Directorios de inclusión adicionales” donde deberás añadir la ruta de la carpeta “include” que has creado antes. Lo siguiente que debes hacer es ir a Vinculador/General y meter la ruta de la carpeta “lib” dentro de “Directorios de bibliotecas adicionales”. Finalmente, dentro de Vinculador/Entrada añadir “glfw3.lib” y “opengl32.lib” en “Dependencias adicionales”. Con esto, ya puedes incluir GLFW en tu proyecto.

2.2.2. GLFW en Linux

En tu sistema Linux, siempre que tengas los drivers gráficos bien instalados, seguramente tengas ya la librería *libGL.so* y simplemente tengas que al linkar el programa usar las flags *-lglfw3 -lGL*. Aunque para evitar problemas con dependencias al compilar con GCC, recomiendo que directamente uses *-lglfw3 -lGL -lX11 -lpthread -lXrandr -lXi -ldl*

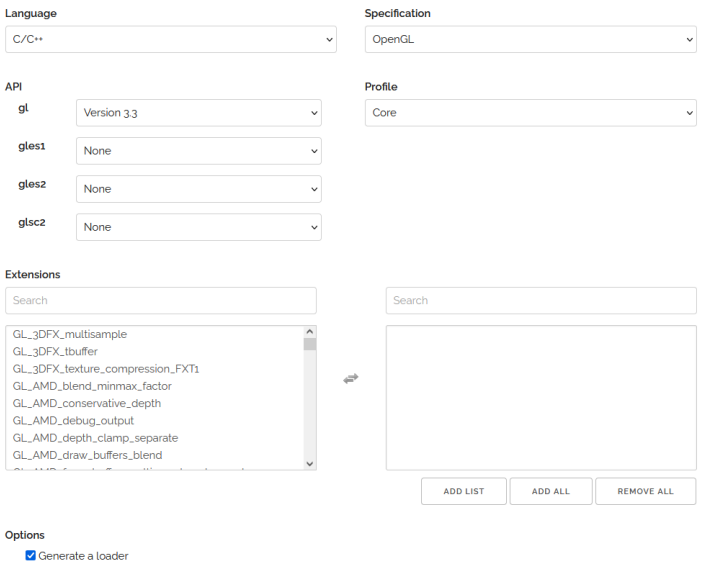
- *Los dos primeros son para las librerías de GLFW y OpenGL. Hay que tener en cuenta que en distribuciones modernas a veces “lglfw3” es referenciado como simplemente “lglfw” por lo que usa la que necesites de las dos.*
- *X11 es el gestor de ventanas de Linux.*
- *Pthread es la biblioteca de POSIX para la programación multihilo (añade soporte a OpenGL).*
- *Xrandr es una extensión de X11 para un manejo más dinámico de ventanas, GLFW lo usa para operar con tu monitor.*
- *Xi es otra extensión, pero esta vez dedicada al input, necesaria para las capacidades de detección de input de GLFW.*
- *Dl (Dynamic Loading) es necesario para cargar extensiones de OpenGL moderno sin necesidad de enlazarlas directamente.*

Para Linux te recomiendo encarecidamente hacer un archivo Makefile para simplificar la compilación del proyecto. Al final del capítulo junto con el código fuente del programa habrá un Makefile ya configurado.

2.3. GLAD

Ya le podemos decir a nuestro sistema operativo que abra una ventana con la especificación de OpenGL, usando los métodos que tenga la librería de OpenGL que esté usando tu dispositivo... Un momento, ¿qué métodos? Si la implementación como tal de esos métodos especificados en OpenGL depende de tus drivers... ¿Cómo sé en qué parte de la memoria de mi dispositivo están? Ahí es donde ayuda la librería de GLAD.

GLAD es un proyecto open source que se encarga de mapear todas las direcciones de memoria de los métodos de OpenGL de tu librería por ti. Si accedes a su portal web[6] y seleccionas el lenguaje C++, la versión de OpenGL igual o superior a 3.3 (Donde se suele considerar que empieza de verdad OpenGL moderno), dejamos el perfil "Core" (sin retrocompatibilidad con versiones viejas y obsoletas) y nos aseguramos de que esté marcado la casilla de generar el loader, podrás generar el código necesario.



Necesitaremos incluir las cabeceras y compilar el archivo *glad.c* junto a nuestro código. En Windows, arrastra el contenido del ZIP en una carpeta GLAD al lado de la de GLFW (no olvides añadir la ruta de la carpeta include en los directorios de inclusión adicionales), por la parte de *glad.c* ve a Explorador de soluciones/Archivos de origen y haces click derecho. Vas a Agregar/Elemento existente y seleccionas el archivo de *glad.c* que estará dentro de src en tu carpeta GLAD. En Linux, con añadir la ruta de tu carpeta de librerías a la hora de compilar y el archivo de *glad.c* a la hora de linkar vas bien.

[6] Web de GLAD

2.4. GLM

La última librería que, de momento, vamos a añadir a nuestro proyecto es GLM. Esta es simplemente una librería matemática en C++ preparada para trabajar con la especificación de OpenGL. Tanto GLAD como GLM no tendrán mucha importancia en este capítulo, pero serán cruciales en los próximos así que cuanto antes los metamos y empecemos a toquetearlos, mejor.

Para poder añadirlo a nuestro proyecto, nos bajaremos la última release del repositorio de GLM[7]. Al abrirla vemos una carpeta llamada "glm", dentro de esta copiamos el contenido de la otra carpeta llamada "glm" y lo añadimos a una nueva carpeta junto con las de GLFW y GLAD, tanto en Windows (no olvides añadir esta carpeta a los directorios de inclusión adicionales) como en Linux el proceso es parecido a estas últimas.

2.5. HORA DE PROGRAMAR

De momento, tenemos todo lo necesario para empezar a construir nuestra nueva ventana. Vamos a empezar a mostrar algo de código que ya va siendo hora. Lo primero de todo, incluir nuestras librerías. SIEMPRE incluye GLAD antes que GLFW, pues este último necesita conocer las cabeceras de GLAD.

```
#include <glad/glad.h>
#include <glfw3.h>
#include <glm.hpp>
```

Con esto ya podemos inicializar GLFW para crear nuestro contexto de OpenGL

```
int main(){
    // Inicio GLFW
    glfwInit();

    // Le decimos a GLFW las veriones de OpenGL compatibles
    // y el perfil que vamos a usar, el core
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE,
      GLFW_OPENGL_CORE_PROFILE);

    // En el caso de MAC, necesitamos activar el
    // enumerado de compatibilidad
    //glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);

    return 0;
}
```

[7] Repositorio GLM

Como vemos lo primero es iniciar GLFW. Después con glfwWindowHint modificamos los metadatos de la configuración de la ventana individualmente. En este caso, le indicamos la versión de OpenGL que vamos a usar. En mi caso, voy a trabajar con 3.3 (el Major es para el número y el Minor para el decimal), pero tú puedes usar la que quieras. También es importante que le indiquemos que sólo estamos usando la versión Core de OpenGL.

Ahora es el momento para compilar. Si ejecutas el programa y no pasa nada, bien, todavía no hemos creado la ventana. Si algo sale mal lo más probable es que hayas cometido algún fallo a la hora de incluir y/o linkar librerías. Si estás en Windows, Visual Studio te chivará en la terminal qué está pasando. Si estás en Linux... bueno, te dejo unas guías[8] que quizá te echen una mano con el proceso.

Llegó el momento de crear nuestra primera ventana, ya era hora. Lo primero que tenemos que hacer es indicar la creación de la ventana con glfwCreateWindow. Hay que tener en cuenta que crear la ventana y decirle a nuestro GLFW "Hey, tú estás trabajando con esta ventana" son dos cosas distintas. Para eso necesitamos usar glfwMakeContextCurrent indicándole el puntero de la ventana. Al acabar el programa destruimos la ventana y cerramos GLFW:

```
// Creamos la ventana con GLFW
GLFWwindow* window = glfwCreateWindow(
    1280,                 // Ancho
    720,                  // Alto
    "Mi Primera Ventana", // Nombre (En UTF-8)
    NULL,                 // Puntero al monitor
    NULL                  // Puntero a otra ventana
                          //   (Para compartir recursos)
);

// Si no se crea la ventana, abortamos y lo decimos por terminal
if(window == NULL){
    std::cout << "Ha fallado la creacion de la
      ventana" << std::endl;
    glfwTerminate();
    return 0;
}

// Le decimos a GLFW que use la ventana creada
glfwMakeContextCurrent(window);

// Destruimos la ventana y cerramos glfw
glfwDestroyWindow(window);
glfwTerminate();
return 0;
```

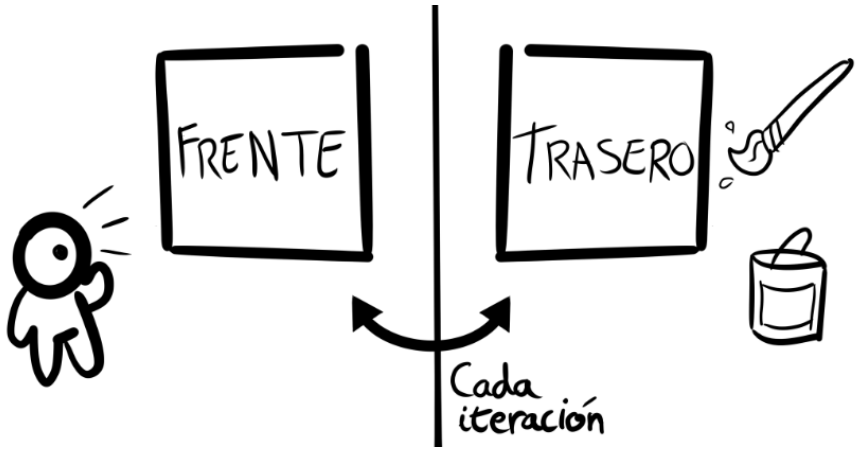
[8] La guía del desarrollador Linux de Red Hat tiene un capítulo de esto: Using Libraries with GCC
También puedes echarle un ojo al vídeo de embeddedarmdev

Ejecútalo. ¿Has visto la ventana? ¿No? Bueno, si eres avispado te habrás dado cuenta que cerramos la ventana tan rápido como la abrimos. ¿Cómo podemos hacer para que la ventana se quede abierta hasta que yo quiera que se cierre? Muy fácil, mantenemos el programa en un bucle condicional apoyándonos en glfwWindowShouldClose (simplemente comprueba si la ventana se ha marcado para cerrar). Este bucle es lo que conoceremos como el bucle de renderizado.

```
...
glfwMakeContextCurrent(window);

// Inicializamos el bucle de renderizado
while(!glfwWindowShouldClose(window)){

}

glfwDestroyWindow(window);
...
```

En cada iteración del bucle nos interesa que OpenGL haga dos cosas: La primera es que termine todos los eventos pendientes por procesar antes de la siguiente iteración (puede parecer que no ya que técnicamente no estamos lanzando ningún evento, pero esto es importantísimo) y la segunda es intercambiar los buffers frontal y trasero.

Por defecto, OpenGL usa un sistema de doble buffer: el frontal (front) y el trasero (back). La idea es que en cada iteración del bucle al usuario le muestres el buffer frontal mientras dibujas el siguiente frame en el buffer trasero. Al acabar la iteración, intercambias los buffers y así evitamos que el usuario vea cosas a medio dibujar o artefactos raros.



```
// Inicializamos el bucle de renderizado
while (!glfwWindowShouldClose(window)) {
    // Procesamos todos los eventos pendientes antes
    // de la siguiente iteracion
    glfwPollEvents();
    // Cambiamos los buffer
    glfwSwapBuffers(window);
}
```

Si todo ha ido bien, al ejecutar el programa deberías tener una ventana completamente oscura y ominosa delante de ti. ¡Has dado tu primer pasito!

2.6. UN POCO DE COLOR

Ahora que ya tienes tu primera ventana, vamos a hacer que haga cosas algo más interesantes que simplemente quedarse mirándote en negro. Pero primero, vamos a iniciar GLAD (que, aunque de momento no lo estemos usando, lo necesitaremos para el futuro)

```
...
glfwMakeContextCurrent(window);

// Inicializamos GLAD con el contexto de GLFW
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {
    std::cout << "Ha fallado la inicializacion
      de GLAD" << std::endl;
    return 0;
}

// Iniciamos el viewport de GLAD con el contexto de OpenGL
glViewport(0, 0, 1280, 720);

// Inicializamos el bucle de renderizado
...
```

Bien, ahora que nos hemos quitado esto de encima vamos a empezar a hacer cosas guays. Primero, vamos a quitarnos de encima al color negro. Para limpiar la ventana con un color en OpenGL podemos usar glClearColor, con cuatro parámetros correspondiente a los colores RGB y al canal Alpha.

Si has sido prematuro y te has aventurado a ejecutar el programa llamando a glClearColor en el bucle de renderizado, te habrás dado cuenta de que todavía sigue negra la pantalla. Recuerda que OpenGL es una máquina de estados, tenemos que indicarle qué buffer está pintando glClearColor. Para eso, llamamos a glClear con el enumerado de buffer de color.

```
// Inicializamos el bucle de renderizado
while (!glfwWindowShouldClose(window)) {
    // Limpiamos la pantalla con el color escogido
    glClearColor(255, 165, 0, 255);
    // Limpiamos el buffer de color
    glClear(GL_COLOR_BUFFER_BIT);

    // Procesamos todos los eventos pendientes antes
    // de la siguiente iteracion
    glfwPollEvents();
    // Cambiamos los buffer
    glfwSwapBuffers(window);
}
```

Con este código debería salirte una bonita ventana naranja. ¿Cómo? ¿Te sale amarilla? Mmh... No puede ser, si el RGB es del naranja. No te asustes, simplemente te has salido del rango con el que trabaja OpenGL en esa función.

Para las funciones que reciben color, OpenGL espera rangos del [0,1]. Si te pasas de estos valores simplemente se fija al máximo o al mínimo dependiendo del caso (clamping), por lo que el 165 del naranja se estaba volviendo un 1, lo que proporcionalmente equivale al 255. Para sacar el valor dentro del rango [0,1], simplemente divide los valores entre 255.

```
//glClearColor(255/255, 165/255, 0/255, 255/255);
glClearColor(1, 0.65, 0, 1);
```

Lo suyo es evitar que en la medida de lo posible el programa haga divisiones ya que son costosas, sobre todo en un bucle, por lo que mejor hazlas tú si puedes

Y voilà, si ejecutas tendrás una pantalla más naranja que... em, una naranja.

2.7. TOMANDO CONTROL

Está bien esto de poder cambiar el color a cualquiera que queramos, pero ¿no estaría mejor poder hacerlo en tiempo de ejecución? Vamos a reclamar algo de poder sobre nuestro pequeño reino virtual y cuadrado.

Para ello lo primero que vamos a hacer es guardar el color en un vector. GLM tiene unos tipos definidos muy útiles, entre ellos hay vectores de distintos tipos y longitudes. Vamos a usar para el color un vector de longitud 4. En total, tendremos 4 vectores 4. Crearemos en principio 3 para los colores que iremos

cambiando y guardaremos uno para saber el color actual que estamos pintando. Eres libre de hacer esta parte de otra forma si te crees más guay.

Para controlar qué botones pulsa el usuario, podemos usar los métodos que nos proporciona GLFW. Primero recuperamos el estado de una tecla usando glfwGetKey (junto con la ventana y el enumerado correspondiente) y después comprobamos si está pulsada o no.

```
// Definimos nuestros colores de ventana fuera del main
glm::vec4 color1(0.78, 0.78, 0.78, 0.78);
glm::vec4 color2(0.6, 0.6, 0.6, 1);
glm::vec4 color3(0.22, 0.22, 0.22, 1);

...

// El color actual de la ventana
glm::vec4 colorActual(color1);

// Inicializamos el bucle de renderizado
while (!glfwWindowShouldClose(window)) {
    // Si aprieta la tecla numérica 1
    if (glfwGetKey(window, GLFW_KEY_1) == GLFW_PRESS) {
        colorActual = color1;
    }
    // Si aprieta la tecla numérica 2
    if (glfwGetKey(window, GLFW_KEY_2) == GLFW_PRESS) {
        colorActual = color2;
    }
    // Si aprieta la tecla numérica 3
    if (glfwGetKey(window, GLFW_KEY_3) == GLFW_PRESS) {
        colorActual = color3;
    }

    // Limpiamos la pantalla con el color actual
    glClearColor(colorActual.x, colorActual.y,
      colorActual.z, colorActual.w);
...
```

La forma de acceder a los datos de un vector de GLM depende de su longitud, pero siempre sigue el esquema XYZW. Es decir un vec2 tendrá elementos vec2.x y vec2.y mientras que un vec3 tendrá vec3.x, vec3.y y vec3.z.

Enhorabuena, has sobrevivido al primer capítulo real. ¿No ha sido tan dificil verdad? Bueno, no puedo prometer nada de los siguientes, pero intentaré que el viaje sea al menos igual de tranquilo. Te dejo un enlace con todo el código fuente de este capítulo[9].

[9] Código fuente Fase 1

3. UNA ESCENA CUBISTA

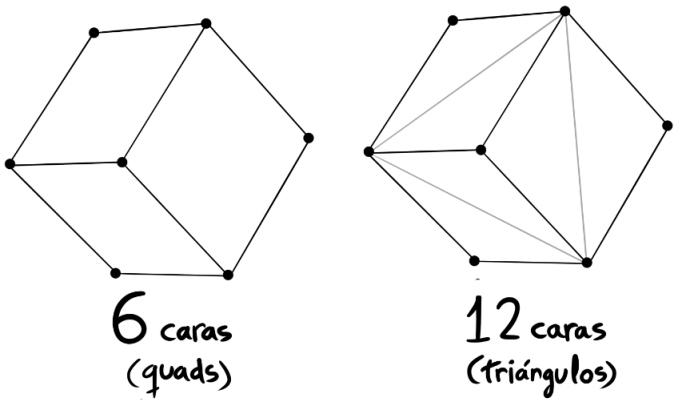
Ya sabes crear tu propia ventana, incluso a pintarla a tu gusto, pero el escenario es un poco triste, vamos a poblarlos con cubos. Sí, cubos, cajas, con sus ocho esquinas y seis caras. Si te parece que va a ser un salto muy grande porque no sabes aún ni dibujar un pixel, tienes buena intuición. ¡Pero ya te advertí de que este libro no iba sobre OpenGL! Pasaré de forma superficial por temas complejos, aunque no te librarás de las matemáticas.

Lo primero será pasar del 2D al 3D. Para lograr esto sólo tenemos que crear la ilusión de profundidad. Pero antes, hay cuatro conceptos clave que tenemos que dominar: Los vértices y triángulos, los shaders, las transformaciones y los buffers. Vamos por partes, primero la teoría.

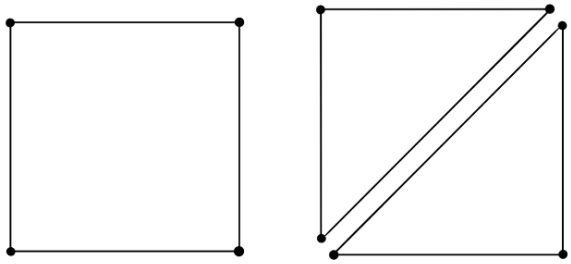
3.1. VÉRTICES Y TRIÁNGULOS

¿Qué es un cubo? ¡Un miserable montón de vértices! Un vértice no es más que la representación de un punto en un espacio tridimensional. En el mundo del renderizado 3D nos gusta mantener las cosas simples. Piensa en cualquier objeto 3D que recuerdes, seguro que está formado por meros triángulos (quizá cientos de miles). Los triángulos son la forma geométrica más simple y eficiente que existe. Y esos triángulos no son más que tres vértices unidos por aristas.

OpenGL sabe eso, por lo que se ha optimizado para dibujar triángulos. Es su especialidad. Incluso si al hacer el modelo has usado quads (un polígono de cuatro lados), lo más probable es que tu gráfica lo transforme en dos triángulos de tres vértices cada uno.



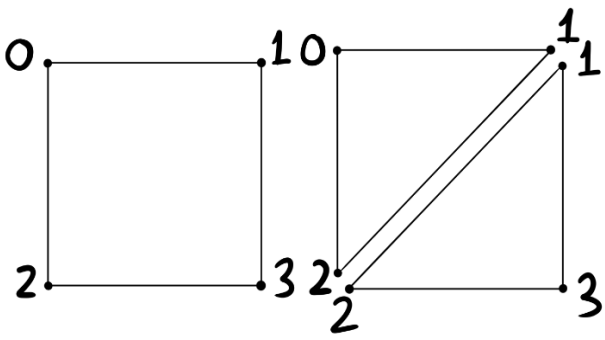
Inocentemente habrás pensado que bueno, aumentan las caras, pero como no aumentan los vértices tampoco es tanto problema. Sin embargo, imagínate que tienes un cuadrado y que vas a dibujarlo usando dos triángulos, pero sólo puedes usar esos cuatro vértices. Usas tres de esos vértices para el primer triángulo y... vaya, sólo te queda uno. Vas a tener que redibujar dos vértices que ya habías dibujado. Por eso, al pasarlo a triángulos, cuatro vértices se convierten en la práctica en seis.



Eso hace que nuestro cubo no tenga ocho esquinas, tenga treinta y seis. No te preocupes, aunque es cierto que dibujar los vamos a tener que dibujar todos, si usamos indexación nos libramos de guardar los vértices repetidos.

La indexación no es más que guardar una lista de índices con el orden de dibujado de los vértices. De esta forma puedes tener sólo cuatro vértices en memoria, ahorrando un buen montón de espacio.

```
// Vértices únicos (4 en lugar de 6)
vertices = [V0, V1, V2, V3]

// Índices, indican el orden de
// dibujado (6 elementos)
indices = [0, 1, 2,  2, 1, 3]
```



Y en cuanto a las aristas... Si dominas los puntos, dominas las líneas. Al fin y al cabo, una línea no es más que una sucesión de puntos muy juntitos.

3.2. AQUEL QUE PINTA: EL SHADER

Los triángulos y los vértices son la estructura que ayuda a dar forma al mundo, el shader es el pintor que le da forma y color. Un shader no es más que un pequeño programa que se ejecuta directamente en tu GPU, seguramente unos miles de veces por escena. OpenGL sigue una pipeline de renderizado específica, y nos da la responsabilidad de programar los shaders a nosotros en tres ocasiones:

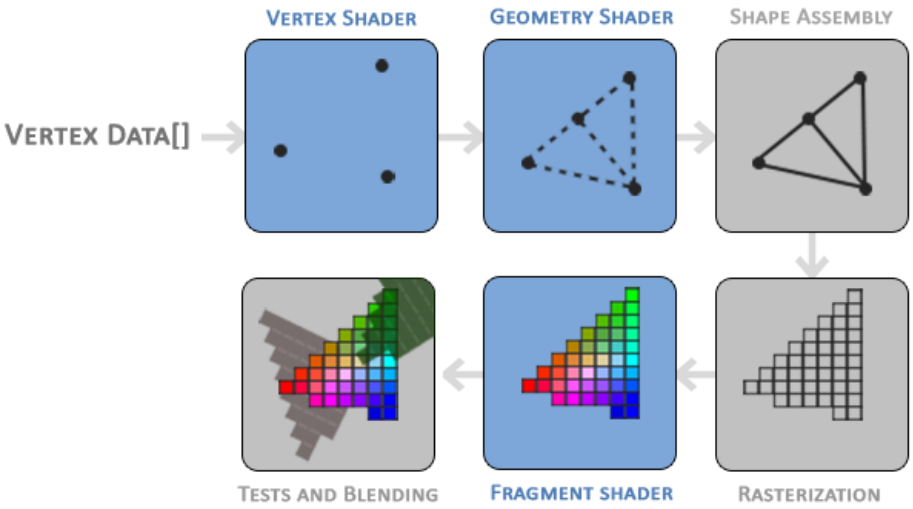


Figura 1: Pipeline Gráfica[10]

Los marcados en azul son aquellos que son programables, en gris los que tu gráfica hace automáticamente por debajo.

3.2.1. Vertex Shader

El Vertex Shader se ejecuta una vez por vértice. Transforma un punto 3D de la escena en un punto 2D de la pantalla, aplica todas las transformaciones de movimiento, rotación y perspectiva para averiguar cómo se debería pintar un objeto en tu monitor.

Piensa en un lápiz, el lápiz está en la mesa, un amigo tuyo y tú estáis observando el lápiz. Ambos estaréis observando el lápiz desde posiciones, ángulos y perspectivas completamente distintas a pesar de que para los dos el lápiz está en la misma posición. El Vertex Shader es el que calcula cómo se ve el lápiz desde tu perspectiva, que sería la pantalla.

[10] Fuente: LernOpenGL - Hello-Triangle

3.2.2. Fragment Shader

El Fragment Shader se ejecuta una vez por fragmento. Un fragmento es un grupo de píxeles (aunque generalmente suele ser de un solo píxel) que nuestra gráfica haya decidido agrupar. Sí, eso significa que se va a ejecutar muchísimas veces. Este programa es el responsable de decidir el color final de cada píxel.

A veces el cálculo es directo. ¿Lo quieres azul? Lo pintas azul. Sin embargo, en la mayoría de casos el cálculo del color es mucho más complejo de lo que parece. ¿Luces? ¿Sombras? ¿Texturas? Todo lo que afecte al color percibido de un objeto lo ha de calcular este programa, lo cual no es moco de pavo.

3.2.3. Geometry Shader

Este es un shader opcional, es decir puedes renderizar objetos complejos sin la necesidad de programarlo. Este se ejecuta justo después del Vertex Shader por lo que recibe todas las formas ya generadas (Los triángulos, líneas o puntos que hayas mandado a dibujar). Su trabajo es generar o eliminar geometrías nuevas teniendo como base las recibidas.

Este programa tiene casos de uso más específicos. Por ejemplo, si quisieses dibujar la silueta de tu modelo 3D podrías hacer que este programa genere líneas gruesas a través del borde del modelo. También es muy usado para sistemas de partículas, de sombras y para herramientas de depuración.

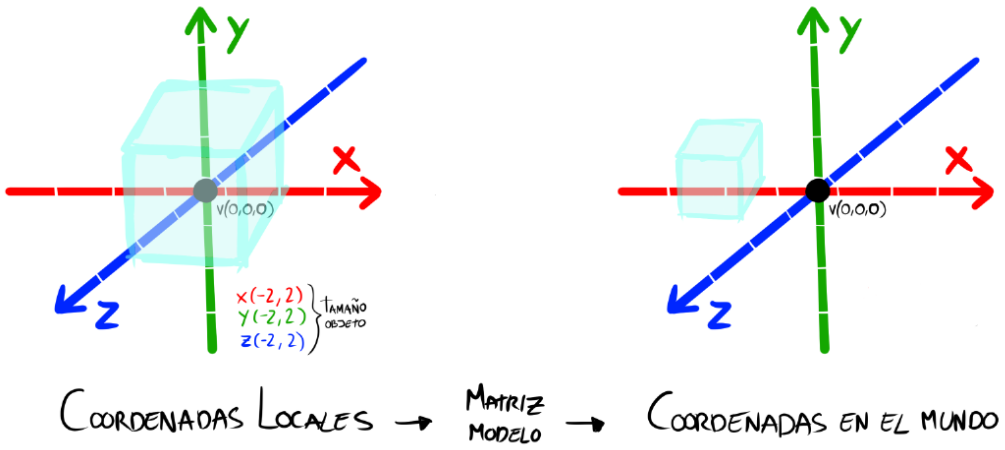
3.3. TRANSFORMANDO EL MUNDO

Hemos mencionado muy alegremente que el Vertex Shader ejecuta las transformaciones necesarias para averiguar dónde pintar un objeto. ¿Pero qué significa exactamente esto? Bueno, aquí es donde entran nuestras amigas las matemáticas.

Volvamos al lápiz en un momento, sabes que el lápiz está posicionado en unas coordenadas específicas del mundo (En los ejes X, Y y Z). Ahora quieres moverlo, pero no vale usar las manos, necesitas calcular las nuevas coordenadas a las que va a pertenecer. Esto es un ejemplo de transformación, y para ello usaremos matrices. Espero que te caigan bien.

3.3.1. Matriz Modelo

Esta es la matriz básica que describe la posición, rotación y escala de un objeto en el mundo. Todo objeto tiene sus coordenadas locales, que suelen empezar con el origen de coordenadas (todos los ejes a 0) en el centro del objeto, esto indica el tamaño del mismo. La matriz modelo pasa esas coordenadas locales a las coordenadas del mundo, es como si "colocase" el lápiz en su sitio.



Es por esto que todo objeto debe tener su matriz modelo y sobre esta se harán las operaciones de traslación, rotación y escalado. Voy a explicar las transformaciones brevemente, ya que, aunque luego no hagamos estos cálculos a mano nosotros, sino que usaremos la librería de GLM, considero que es necesario que al menos la teoría te suene.

3.3.1.1. Traslación

El proceso de traslación no es más que añadirle otro vector al vector de posición. Es decir, tienes tu lápiz en el (P_1, P_2, P_3) y lo quieres mover dos puntos en el eje Y (0, 2, 0). La posición entonces se define por el vector (P_1, P_2+2, P_3).

Teniendo que usar matrices, si ponemos ese segundo vector en los primeros tres valores de la última columna de una matriz 4x4 y el resultado será idéntico al anterior propuesto.

$$\begin{bmatrix} 1 & 0 & 0 & P_1 \\ 0 & 1 & 0 & P_2 \\ 0 & 0 & 1 & P_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + P_1 \\ y + P_2 \\ z + P_3 \\ 1 \end{pmatrix}$$

Para estos cálculos, partimos siempre de la matriz identidad al ser un elemento completamente neutro. Además, ajustamos el vector para añadir la coordenada homogénea. Actualmente sólo nos sirve para poder hacer más operaciones, pero más adelante será mucho más útil.

3.3.1.2. Rotación

Una vez explicada la traslación, la rotación es muy similar. Todo depende del eje en el que quieras aplicar la rotación. Construyes la matriz con el ángulo correspondiente y lo multiplicas con el vector posición de tu objeto para calcular la nueva posición del objeto rotado.

Estos ángulos deberán estar en radianes ya que usaremos tanto el coseno como el seno a la hora de hacer los cálculos.

- Eje X:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sen\theta & 0 \\ 0 & sen\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ cos\theta \cdot y - sen\theta \cdot z \\ sen\theta \cdot y + cos\theta \cdot z \\ 1 \end{pmatrix}$$

- Eje Y

$$\begin{bmatrix} cos\theta & 0 & sen\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sen\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} cos\theta \cdot x + sen\theta \cdot z \\ y \\ -sen\theta \cdot x + cos\theta \cdot z \\ 1 \end{pmatrix}$$

- Eje Z

$$\begin{bmatrix} cos\theta & -sen\theta & 0 & 0 \\ sen\theta & cos\theta & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} cos\theta \cdot x - sen\theta \cdot y \\ sen\theta \cdot x + cos\theta \cdot y \\ z \\ 1 \end{pmatrix}$$

*A pesar de para la mayoría de funciones necesitarás el ángulo en radianes y no en grados, es fácil pasar de uno a otro sabiendo que ángulo = radianes * (180/PI), que también se puede expresar como radianes = ángulo * (PI/180)*

3.3.1.3. Escalado

El escalado es simplemente incrementar la longitud del vector de dirección por cualquier valor que queramos. Es un aumento de magnitud.

$$\begin{bmatrix} P1 & 0 & 0 & 0 \\ 0 & P2 & 0 & 0 \\ 0 & 0 & P3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} P1 \cdot x \\ P2 \cdot y \\ P3 \cdot z \\ 1 \end{pmatrix}$$

3.3.1.4. Combinación de matrices

El potencial que contienen las matrices se ve claramente cuando comprobamos que podemos hacer distintas transformaciones a la vez si hacemos una multiplicación de matiz x matriz.

Lo único que hay que tener en cuenta es que esta operación no es conmutativa, por lo que el orden importa. Es decir, no es lo mismo A x B que B x A. Para evitar que alguna transformación se aplique mal, es importante que el orden sea escalado, rotación y traslación. Ten en cuenta que el orden en el que se aplica la multiplicación es de derecha a izquierda.

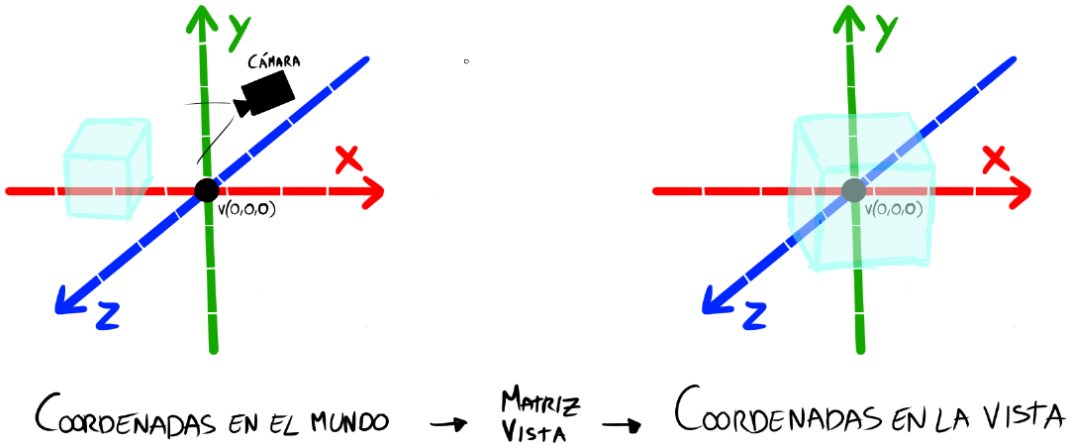
Pongamos el ejemplo de que quieres escalar el vector del lápiz por 2 y al mismo tiempo moverlo siguiendo el vector (2, 4, 6). La matriz de transformación sería la siguiente:

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 4 \\ 0 & 0 & 1 & 6 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 2 \\ 0 & 2 & 0 & 4 \\ 0 & 0 & 2 & 6 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Aplicando esta matriz, podrás hacer ambas transformaciones de una. ¡Magia!

3.3.2. Matriz Vista

Si la matriz modelo colocaba el lápiz en el mundo, la matriz vista lo recoloca teniendo en cuenta un punto de vista. Por ejemplo, dónde está el lápiz desde nuestra perspectiva. También usa la traslación, rotación y escalado.

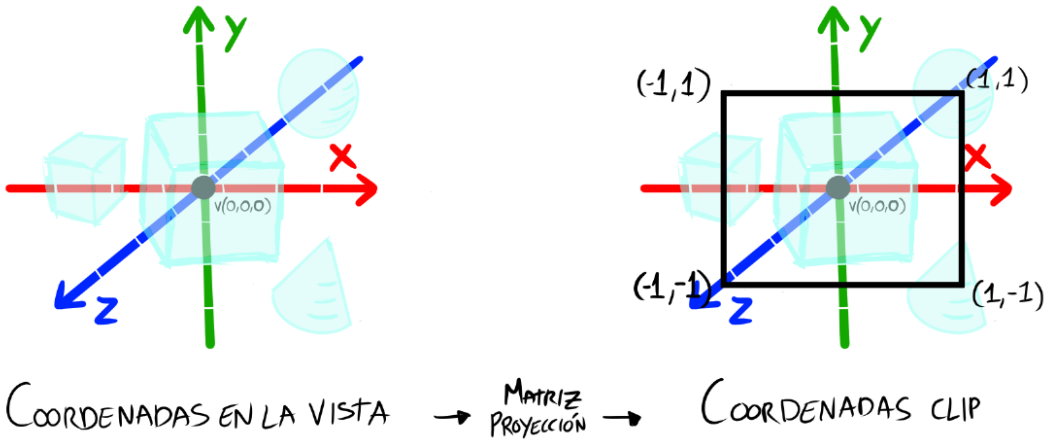


Como ves el proceso es primero colocar el lápiz en el mundo, luego recolocarlo según la vista desde la que queramos observar (recuerda que puedes tener un montón de cámaras y, por lo tanto, un montón de vistas).

3.3.3. Matriz Proyección

La viewport final de OpenGL, es decir la ventana que nosotros vemos, trabaja en rangos de [-1, 1] para después transformar cada punto en coordenadas de píxeles. Cualquier cosa fuera de este rango simplemente no se mostrará en pantalla. Ahora ya tenemos nuestro lápiz colocado según nuestra perspectiva después de haber usado la matriz modelo y vista, ahora vamos a transformar los puntos a este rango.

Para poder obtener ese rango, primero usaremos la matriz proyección para calcular qué partes se van a ver en nuestra pantalla y qué no (A estas nuevas coordenadas se les conoce como coordenadas clip) y después dividiremos por la coordenada homogénea para obtener ese rango de [-1, 1].



Para calcular las coordenadas clip, aplicamos las transformaciones necesarias para obtener uno de los dos tipos de proyección.

3.3.3.1. Ortogonal

La proyección ortogonal es aquella en la que los objetos mantienen siempre su tamaño real, haciendo que sus líneas paralelas lo sean tanto en el mundo 3D como en nuestra pantalla.

Esto hace que objetos lejanos se vean igual que objetos cercanos y que técnicamente no haya un punto de fuga. Si tienes dos lápices, uno al fondo del pasillo y otro al frente, ambos se verán del mismo tamaño. La vista isométrica, tan utilizada por los RTS, es un tipo de proyección ortogonal, por ejemplo.

3.3.3.2. Perspectiva

La proyección perspectiva es la que más se acerca a cómo funciona el ojo humano. Los objetos cercanos se verán más grandes que los lejanos y todas las paralelas convergerán en un punto de fuga.

Esta proyección aporta un efecto de profundidad y volumen, es la más típicamente usada. Esta proyección necesita que le informes cuatro cosas:

- Campo de visión (FOV): El ángulo que abarca tu visión.
- Relación de aspecto: La relación entre el ancho y el alto. Por ejemplo, en una pantalla de 1920x1080p la relación de aspecto es 1,7 (1920/1080 ~ 16/9)

- Plano cercano: Lo más cerca que puede ver tu visión, si un objeto se encuentra más cerca, desaparece.
- Plano lejano: Lo más lejos que puede ver tu visión, si algo está más lejos de aquí no se dibujará.

3.4. BUFFERS, EL ALMACÉN

He dejado lo más sabroso para el final. Si tú guardas un vector con por ejemplo una posición de un cubo en tu programa, tu PC automáticamente lo almacenará en la RAM. Bien, ¿no? Pues no, fatal. Quien dibuja no es tu CPU sino tu GPU, el cual usa su propia memoria, la VRAM. El problema es que para guardar cosas en la VRAM vamos a tener que trabajar un poquito más que para la RAM. Para eso están los buffers, que no son más que trozos de memoria que reservamos de la gráfica para guardar lo que queramos.

3.4.1. VBO: Vertex Buffer Object

Este es el más simple. ¿Qué guarda? Números, en general, ahí a piñón. Pueden ser coordenadas, colores, texturas... Lo que te apetezca y necesites.

```
// Preparamos una variable para la dirección de memoria del VBO
unsigned int VBO;
// Pedimos a OpenGL que nos dé un VBO nuevo y guardamos
// su dirección de memoria
glGenBuffers(1, &VBO);
// Enlazmamos el buffer. Es como decir "OpenGL, a partir de
// ahora, cuando hable del GL_ARRAY_BUFFER me refiero a este VBO"
glBindBuffer(GL_ARRAY_BUFFER, VBO);
// Le decimos los datos que va a tener el buffer
glBufferData(
    GL_ARRAY_BUFFER,         // Le indicamos que va a ser un VBO
    sizeof(vertices_cubos), // El tamaño que va a tener
    vertices_cubos,          // Le pasamos el array que contiene
                             //  los datos
    GL_STATIC_DRAW           // Le indicamos que va a ser
                             //   estático, ya que no se va a
                             //   modificar una vez guardado
);
```

Que le indiques a OpenGL si se van a modificar los datos (GL_DYNAMIC_DRAW) o si no (GL_STATIC_DRAW) sirve para que él por debajo optimice la memoria según el caso

3.4.2. EBO: Element Buffer Object

Antes mencionamos que para dibujar los triángulos si guardamos los índices podemos ahorrar memoria, pues aquí es donde los guardamos.

```
// Creamos el EBO y guardamos la dirección de memoria,
// como antes
unsigned int EBO;
glGenBuffers(1, &EBO);
// Enlazamos
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
// Guardamos
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices_cubos),
      indices_cubos, GL_STATIC_DRAW);
```

Este simple truco hará que pasemos de tener tres vértices por triángulo a dos o incluso uno. Piensa en los millones de triángulos que dibujarás cada frame y verás por qué es tan importante.

3.4.3. VAO: Vertex Array Object

Este es el que coordina un poco el cotarro. Es decir, el que indica cómo estás organizando la información de tus VBOs. Por ejemplo, imagina que quieres guardar en el buffer no sólo la posición sino el color de un objeto, en el VBO estarán todos metidos en fila pues el VAO es el que le dice a OpenGL “Los primeros 3 floats son de la posición, los otros 4 del color y así sucesivamente”. Sin el VAO se lo tendrías que decir tú a OpenGL cada vez que dibujases.

```
// Generamos y enlazamos el VAO para configurarlo
unsigned int VAO;
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);

// Enlazmos el VBO
glBindBuffer(GL_ARRAY_BUFFER, VBO);
// Ahora es donde configuramos el VBO
glVertexAttribPointer(
    0,                  // Le decimos que vamos a ajustar desde
                        //   la primera posición (0)
    3,                  // Que va a tener tamaño 3 (lo vamos a
                        //   usar para guardar la posición del cubo)
    GL_FLOAT,           // Que los datos van a ser de tipo float
    GL_FALSE,           // Que los datos no necesitan ser normalizados
    3 * sizeof(float),  // El tamaño total del buffer (3 floats)
    (void*)0            // Y el puntero desde el que empieza
                        //   (como no tenemos ningún offset, el 0)
);
//Activamos la posición 0 que es la que acabamos de ajuntar
glEnableVertexAttribArray(0);
// Guardamos también el EBO, así lo relacionamos con el VBO
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);

// Cuando terminamos de configurar, podemos desenlazar el VAO
glBindVertexArray(0);
```

Piensa en el VAO como un perfil que guarda la configuración de un VBO y un EBO específicos. De forma que en el bucle de renderizado con cambiar de VAO cambias los datos y cómo OpenGL los está interpretando. Esto va a ser útil cuando tengas que dibujar 100 modelos diferentes, no sólo un par de cubos.

¿Lo has pillado? VBO para los datos, EBO para los índices y el VAO para cómo estos están organizados. Parece complejo pero una vez lleguemos a la programación toda la teoría de este y los otros puntos se verán con más claridad. Y hablando de llegar a la programación...

3.5. A PINTAR CUBOS

Ya hemos aprendido que nuestros cubos no son más que vértices en el espacio, que estos vértices se agruparán en triángulos, dónde los guardamos en la GPU, hemos visto cómo colocar esos triángulos desde nuestra perspectiva y hasta sabemos cómo los vamos a pintar. Ahora toca por fin ponernos a programar un poquito, ¿no te parece?

Lo primero, para las matrices y los punteros, necesitaremos importar dos extensiones de GLM que ya te trajiste cuando te bajaste la librería sólo que son opcionales así que no se importan por defecto.

```
#include <glm.hpp>
#include <gtc/matrix_transform.hpp>
#include <gtc/type_ptr.hpp>
```

Una cosa a la que quiero que te acostumbres es a definir tus propios tipos. Vas a hacer tu propio motor gráfico, ¡tendrán que seguir tus normas así que aprovéchalo! Para ir calentando, de momento vamos a crear el tipo Color que tendrá el valor RGB y el canal Alpha. También definimos el operador de igualdad.

```
// Definimos nuestro propio tipo color
typedef struct Color {
    float r, g, b, a;
    bool operator==(const Color& c) const {
        return (
            r == c.r &&
            g == c.g &&
            b == c.b &&
            a == c.a
            );
    }
} Color;
```

3.5.1. Creando los shaders

Ahora vamos a bajar los Shader un poco de la teoría a la práctica. Los programas son muy sencillos, pero hay conceptos que debemos dominar. Vamos a empezar por el Fragment Shader, ya que es el más fácil... de momento.

```
#version 330 core

uniform vec4 ourColor;
out vec4 FragColor;

void main(){
    FragColor = ourColor;
};
```

Como puedes ver es un programa muy sencillo, GLSL (el lenguaje que usa OpenGL para los shaders) es bastante similar a C. Sin embargo, seguro que te surgen algunas dudas al leerlo.

Primero, el comentario con la versión, es completamente obligatorio. Esta línea es la que le indica al compilador con qué versión estás trabajando. En caso de no informarlo asumirá la versión más antigua posible, lo cual seguramente haga que explote alguna cosa o dos.

Lo segundo. ¿Uniform? ¿Out? ¿Qué tipos son esos? Es muy sencillo en verdad. Piensa en el uniform como una variable global. Tu GPU no puede acceder como si nada a variables que hayas instanciado en tu programa, osea en la CPU. Un uniform conecta ambas partes permitiendo su acceso completo. Esta forma de trabajar añade la ventaja de que, aunque en tu programa cambie el valor del uniform, mientras se está dibujando en la GPU se mantiene el mismo evitando que se generen artefactos. Para informar ese vec4 desde nuestro programa deberemos usar glUniform4f (lo de 4f porque es un vector de 4 floats, si fuese un int sería 1i, por ejemplo).

```
// Informamos el uniform, pero primero lo localizamos
// con la dirección del shader y el nombre del uniform
glUniform4f(
    glGetUniformLocation(ID_Shader, Nombre_Uniform),
      valor1, valor2, valor3, valor4
);
```

Por último, el tipo Out (También existe el In) es para pasar información entre shaders. ¿Recuerdas que esto es un pipeline y que los shaders van en orden? Bien, pues si el Vertex Shader (el primero) le quiere pasar información al Fragment Shader tiene que enviarlo en una variable Out, mientras que el

Fragment Shader tendrá que recibir una variable del mismo tipo y nombre, pero marcada como In.

```
// Desde el Vertex Shader lo declaramos como Out
out vec3 ejemplo;

void main() {
    ejemplo = vec3(1.0, 0.0, 0.0);
}

// ------------------------------------------

// Desde el Fragment Shader lo declaramos como In
in vec3 ejemplo;
// Este es especial, es el color final del píxel
out vec4 FragColor;

void main() {
    FragColor = vec4(ejemplo, 1.0);
}
```

El Vertex Shader no tiene por qué tener un Out definido, pero el Fragment Shader está obligado a tener por lo menos el Out de un vec4 llamado FragColor, que es el color final del píxel.

Pero si recuerdas bien, el Vertex Shader trabaja a nivel de vértice mientras que el Fragment a nivel de fragmento (píxeles). Entonces, si le digo que un vértice es azul y el siguiente es rojo, ¿cómo sabe el Fragment Shader cómo pintar los píxeles intermedios? Lo que hace OpenGL es interpolar los valores automáticamente, es decir en el resultado final verás un degradado de azul a rojo.

Ahora que hemos dejado esto claro, vamos al Vertex Shader que con todo lo explicado va a ser también sencillo de comprender.

```
#version 330 core
layout (location = 0) in vec3 aPos;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main(){
      gl_Position = projection * view * model
            * vec4(aPos, 1.0f);
};
```

A simple vista puedes ver las tres matrices de transformación multiplicándose en el orden que mencionamos antes (proyección * vista * modelo) y multiplicando el resultado por el vector posición (añadiendo la coordenada homogénea). Si te fijas en cómo está declarado este vector posición, se puede ver que lo obtenemos del layout, es decir del buffer que hayamos preparado (el VBO) y de la posición 0 que previamente habremos configurado (en el VAO). Como es algo que el shader recibe, lo marcamos como In.

Lo último que queda explicar es ese gl_Position. Es variable que define la posición del vértice actual que está dibujando. Piensa en él como en una variable global que se puede modificar en cualquier momento de la ejecución de los shaders. Por eso el Vertex Shader no necesita ni declararlo ni pasárselo al siguiente shader.

Ya tenemos nuestro Vertex Shader y nuestro Fragment Shader. Muy sencillitos. Lo importante ahora, ¿cómo hacemos que nuestro programa lo lea? Pues tienes dos opciones. La que menos te recomiendo, sobre todo pensando en que este programa será algún día tu motor gráfico, es guardarlo como una cadena de texto en el programa.

```
const char *fragmentShaderFuente =
    "#version 330 core\n"
    "uniform vec4 ourColor;\n"
    "out vec4 FragColor;\n"
    "void main()\n"
    "{\n"
    "   FragColor = ourColor;\n"
    "}\n\0";
```

Esta forma es conveniente y rápida para un programa sencillo, pero no planeamos hacer algo sencillo en esta casa. Necesitarás que el usuario, si quiere, pueda añadir, quitar y modificar los shaders a su conveniencia y, por lo tanto, como motor gráfico, deberías ser capaz de leer archivos de texto y entenderlos como tus shaders. Por eso, vamos a crear nuestra propia clase shader que leerá un archivo y hará todo lo necesario para que OpenGL lo use como suyo. No iba en broma cuando dije que necesitabas saber sobre programación orientada a objetos.

Antes de nada, vamos a desgranar qué deberá hacer esta clase shader. Lo primero leer el archivo, lo segundo es dárselo de comer a OpenGL. No voy a entrar mucho en detalle en lo primero, al fin y al cabo, leer un archivo de texto es más un ejercicio de C++ que de programación de gráficos.

```
// Seteamos las variables iniciales de almacenamiento y lectura
string vertexCode{}, fragmentCode{};
ifstream vShaderFile{}, fShaderFile{};
// Permitimos a los ifstream lanzar excepciones
vShaderFile.exceptions (ifstream::failbit | ifstream::badbit);
fShaderFile.exceptions (ifstream::failbit | ifstream::badbit);
try{
    // Abrimos los archivos
    vShaderFile.open(rutaArchivoVertexShader);
    fShaderFile.open(rutaArchivoFragmentShader);
    // Creamos as variables para el flujo de datos
    stringstream vShaderStream{}, fShaderStream{};
    // Cargamos el buffer de los archivos en los flujos
    vShaderStream << vShaderFile.rdbuf();
    fShaderStream << fShaderFile.rdbuf();
    // Cerramos los archivos
    vShaderFile.close();
    fShaderFile.close();
    // Convertimos el flujo de datos en strings
    vertexCode = vShaderStream.str();
    fragmentCode = fShaderStream.str();
}catch(ifstream::failure e){
    cout << "ERROR::SHADER::FILE_NO_READED" << endl;
}
// Lo pasamos a cadenas de char y nos llevamos el puntero donde
empiezan
const char* vShaderCode = vertexCode.c_str();
const char* fShaderCode = fragmentCode.c_str();
```

Una vez tenemos ya los archivos leídos y con punteros a donde empiezan los programas en memoria (leído todo en una sola línea, prácticamente idéntico a cómo los teníamos en el ejemplo que he propuesto antes) pasamos a la segunda parte, cargarlos en OpenGL. Es muy directo, sólo tenemos que compilar el texto leído.

El proceso es muy similar a cómo hemos creado los buffers, guardando la dirección del espacio que ha creado OpenGL en memoria e informándole de la información que añadimos. Otra cosa con la que quiero que te familiarices aprovechando este momento es el generar un log cuando hay errores.

Ya lo hicimos cuando creamos la ventana, pero es importante mencionar que siempre que leas o llames a una función que te devuelve algo, des por hecho de que quizá no ha funcionado y pintes un log con el error. Esto es importante para cualquier desarrollo, pero para una librería lo es aún más. ¿Te ha pasado que alguna vez un programa que usas fallase y ni siquiera te dice por qué exactamente? No seas tú también ese programa.

```
// Creamos una variable para la direccion de memoria del shader
unsigned int vertex;
// Dos variables, una para saber si ha cargado y otra
// donde guardaremos el log en el caso de que no
int success;
char infoLog[512];

// Inicializamos el VS usando la variable GL_VERTEX_SHADER
vertex = glCreateShader(GL_VERTEX_SHADER);
// Le decimos cual es el codigo (el que hemos leido antes)
glShaderSource(
    vertex,       // La dirección de memoria del VS
    1,            // Número de elementos a crear (pasamos todo
                  //   el código como una linea)
    &vShaderCode, // Dirección de memoria del código leído
    NULL          // Longitud de la línea (al pasárselo null
                  //   da por hecho que el string termina en null)
);
// Compilamos
glCompileShader(vertex);
// Miramos su status para ver si la compilacion ha sido exitosa
glGetShaderiv(vertex, GL_COMPILE_STATUS, &success);
// Si no lo ha sido, obtenemos el Log de error y lanzamos un mensaje
if(!success){
    glGetShaderInfoLog(vertex, 512, NULL, infoLog);
    cout << "ERROR::SHADER::VERTEX::COMPLIATION_FAILED"
      << infoLog << endl;
}
```

El proceso es idéntico para el Fragment Shader (recuerda rehusar las variables success e infoLog). Una vez tengamos ambos cargados, necesitamos linkar ambos en lo que se conoce como un programa de shader. Esto es lo que realmente lee OpenGL y lo que necesitas para empezar a trabajar.

```
// Creamos el Shader Program
ID_Programa = glCreateProgram();
// Le enlazamos el VS y el FS
glAttachShader(ID_Programa, vertex);
glAttachShader(ID_Programa, fragment);
// Linkamos el programa
glLinkProgram(ID_Programa);
// Comprobamos que haya salido bien
glGetProgramiv(ID, GL_LINK_STATUS, &success);
if(!success){
    glGetProgramInfoLog(ID, 512, NULL, infoLog);
    cout << "ERROR::SHADER::PROGRAM::LINKER_FAILED"
      << infoLog << endl;
}
// Limpiamos la memoria (Importante, eh)
glDeleteShader(vertex);
glDeleteShader(fragment);
```

Si el proceso te parece algo tedioso, imagina tener que hacer esto cada vez que cambiases o actualizases los shaders que usa tu motor. Por eso vamos a crear nuestra propia clase Shader que haga este trabajo por nosotros, y más importante, por el usuario.

```
// Importante para asegurarnos de que sólo se importe una vez
#pragma once
// Necesitaremos GLAD para trabajar con OpenGL
#include <glad/glad.h>

// Todas las librerías necesarias para leer el código
#include <string>
#include <fstream>
#include <sstream>
#include <iostream>

struct Shader{
    // ID del programa de shader
    unsigned int ID;

    // Constructor con el path del código del VertexShader
    // y el FragmentShader
    Shader(const char* vertexPath, const char* fragmentPath);

    // Función que activará o desactivará el shader
    void use();

    // Funciones para declarar uniforms, siéntete libre de añadir más
    void setBool(const std::string &name, bool value) const;
    void setInt(const std::string &name, int value) const;
    void setFloat(const std::string &name, float value) const;
    void set4Float(const std::string &name, float value1,
        float value2, float value3, float value4) const;
    void setMatrix4fv(const std::string &name,
        const GLfloat *matrix) const;
};
```

Esta será nuestra cabecera Shader.hpp. Después en nuestro programa Shader.cpp implementaremos el constructor que ejecute la secuencia que antes hemos mencionado: Lee el código de los shaders (las rutas las recibe por parámetro), carga, compila el código leído y linka ambos en un shader program.

Además, para mejorar las capacidades de nuestra clase Shader, vamos a añadirle una función "usar" y unas funciones para cargar uniforms en los shader.

```
void Shader::use(){
    glUseProgram(ID);
}

// Como parámetro recibe el nombre del uniform y el valor
void Shader::setBool(const string &name, bool value) const{
    glUniform1i(glGetUniformLocation(ID, name.c_str()),
        static_cast<int>(value));
}
```

Como puedes ver en los ejemplos, estos métodos son sencillos, de una línea, pero explican muy bien la filosofía que tienes que tener como desarrollador de un motor gráfico: Tu objetivo es abstraer, hacer sencillo lo complicado. No es lo mismo pedirle a tu usuario que haga "glUseProgram(Shader.getID())" que "tuShader.use()".

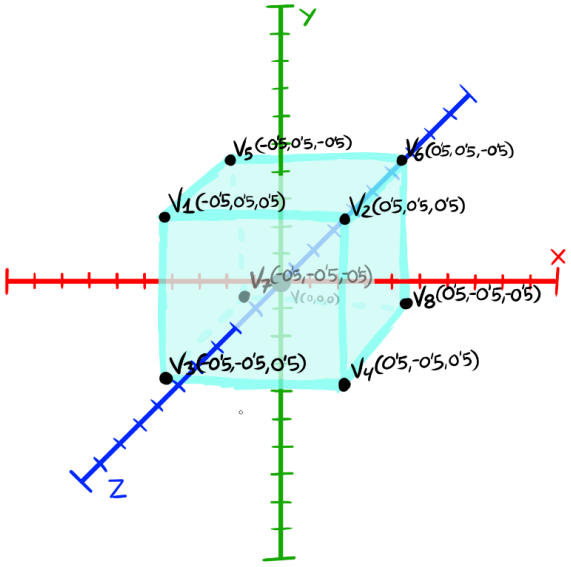
Te dejo el código de la cabecera[11] y la implementación[12] de nuestra clase Shader para que estemos en la misma página.

> *Este apartado ya subía un poco la dificultad, así que no he querido complicarme más, pero* `glGetUniformLocation` *busca dentro del shader el uniform por comparación de cadenas. Esto es un poco costoso, por lo que teniendo en cuenta que va a estar en el bucle de renderizado lo suyo sería que la primera vez que buscase, la clase Shader guardase esa dirección en un hash map, por ejemplo, y luego simplemente lo recuperase cada iteración. Eres libre de añadirlo si quieres.*

3.5.1. Creando las cajas

Ya tenemos los shaders configurados, el pintor tiene sus pinceles listos, ahora vamos a llenarle los buffers de pintura. Pensemos un segundo en cuántos vértices tiene un cubo. Como ya hablamos en su momento, esas 8 esquinas se van a convertir en 36 a la hora de dibujar, porque dividimos todo en triángulos, pero gracias a los índices podemos rehusar los vértices iguales para no cargar demás en memoria.

Entonces, ¿cuántos vértices de verdad tenemos que cargar en memoria? Pues 8. Sí, hemos vuelto al principio, pero es importante que te lo haya explicado de esta forma para que entendieses la triangulación, por qué son necesarios los índices y por qué, después, le diremos a OpenGL que dibuje 36 vértices y no 8. Supongamos un cubo de 1x1, si tenemos el centro de las coordenadas locales en el centro del cubo, eso significa que se extenderá 0.5 en todos sus ejes.



[11] Archivo Shader.hpp
[12] Archivo Shader.cpp

```
float vertices[] = {
    // Cara delantera (Eje Z = 0.5)
    -0.5f,  0.5f,  0.5f,  // Vértice 0 (Esquina superior izquierda)
     0.5f,  0.5f,  0.5f,  // Vértice 1 (Esquina superior derecha)
    -0.5f, -0.5f,  0.5f,  // Vértice 2 (Esquina inferior izquierda)
     0.5f, -0.5f,  0.5f,  // Vértice 3 (Esquina inferior derecha)

    // Cara trasera (Eje Z = -0.5)
    -0.5f,  0.5f, -0.5f,  // Vértice 4
     0.5f,  0.5f, -0.5f,  // Vértice 5
    -0.5f, -0.5f, -0.5f,  // Vértice 6
     0.5f, -0.5f, -0.5f   // Vértice 7
};
```

Ahora vamos con los índices, para ellos nos imaginamos las caras partidas en triángulos como en el ejemplo del apartado 3.1. Es un poco complejo si no tienes mucha capacidad espacial, sin embargo es importante que aproveches y hagas el esfuerzo de intentan entender cara por cara cómo escoges los índices ahora que la complejidad, aunque pueda asustar, es bajita.

```
// El orden de las caras en el cubo
unsigned int indices[] = {
    // Cara delantera
    3, 2, 0,
    3, 0, 1,

    // Cara trasera
    7, 6, 4,
    7, 4, 5,

    // Cara izquierda
    7, 5, 1,
    7, 1, 3,

    // Cara derecha
    6, 4, 0,
    6, 0, 2,

    // Cara inferior
    7, 7, 2,
    7, 2, 6,

    // Cara superior
    5, 4, 0,
    5, 0, 1
};
```

Esta es la potencia de los índices, con sólo 8 vértices en memoria estamos dibujando 36. Imagina cuánto estaríamos ahorrando si en vez de un cubo dibujásemos un modelo complejo.

Ahora vamos a inicializar los buffers que son donde va a ir toda esta información, que la GPU si no, no se entera.

```
// Iniciamos nuestros Buffers y el Array Buffer
unsigned int EBO, VBO, VAO;
glGenBuffers(1, &EBO);
glGenBuffers(1, &VBO);
glGenVertexArrays(1, &VAO);

// Enlazamos el Array para indicarle a OpenGL que estamos
trabajando ahora en él
glBindVertexArray(VAO);

// Enlazamos el Vertex Buffer sobre el Array Buffer
glBindBuffer(GL_ARRAY_BUFFER, VBO);

// Le indicamos el tamaño del buffer
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);
// 3 Floats para las posiciones de cada vértice
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
      3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices),
      indices, GL_STATIC_DRAW);
```

Perfecto, ya tenemos los vértices, los índices y le hemos dado la información a OpenGL. Ya estamos casi casi.

3.5.2. Creando la escena

Ahora que tenemos las cajas y los shaders, vamos a construir la escena. ¿Qué te parece si empezamos poniendo una cajita en el mundo? Luego metemos más, no te preocupes. Lo primero de lo primero es preparar las matrices modelo, vista y proyección.

```
// Creamos la matriz modelo, no necesita alteraciones
glm::mat4 model = glm::mat4(1.0f);

// Creamos la matriz vista
glm::mat4 view = glm::mat4(1.0f);
// Colocamos la "Cámara" con la que vamos a ver en el mundo
// Usamos la función de GLM para hacer la transformación
//   de traslación
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));

// Creamos la matriz de proyeccion
glm::mat4 projection{};
// Usamos la función de GLM para la proyección perspectiva
projection = glm::perspective(
    glm::radians(45.0f),              // Ángulo
    static_cast<float>(ANCHO / ALTO), // Relación de aspecto
    0.1f,                             // Plano Near
    100.0f                            // Plano Far
);
```

Bien, antes de entrar en el bucle de renderizado vamos a darle un colorcito al fondo y a instanciar nuestra clase Shader con los shaders que creamos antes. El nombre y la extensión de los archivos no son realmente relevante, siempre que sea texto.

```
// Damos un color base al fondo para que no salga negro puro
glClearColor(0.2f, 0.3f, 0.3f, 1.0f);

// Cargamos nuestros shaders basicos
Shader ourShader("./Shaders/vertex_shader_basico.vs",
      "./Shaders/fragment_shader_basico.fs");
```

Finalmente, iniciamos nuestro bucle de renderizado. Lo primero del bucle, recuerda, que sea borrar el buffer de color. Lo último también ha de ser lo de terminar todos los eventos pendientes y cambiar el buffer frontal por el trasero. La magia ocurrirá en medio. Le diremos a OpenGL que queremos usar nuestro programa de shader, informamos los uniforms y usamos glDrawElements para al fin dibujar nuestro cubo.

```
// Inicializamos el bucle de renderizado
while (!glfwWindowShouldClose(window)) {
    // Tenemos que limpiar el color en cada iteración
    glClear(GL_COLOR_BUFFER_BIT);
    // Le indicamos a OpenGL que vamos a usar nuestro shader
    ourShader.use();
    // Metemos las matrices en los uniforms
    ourShader.setMatrix4fv("model", value_ptr(model));
    ourShader.setMatrix4fv("view", value_ptr(view));
    ourShader.setMatrix4fv("projection", value_ptr(projection));
    // Metemos el color, en este caso rojo
    ourShader.set4Float("ourColor", 1.0, 0.0, 0.0, 1.0);

    // Dibujamos el cubo
    glDrawElements(
        GL_TRIANGLES,    // El tipo de primitiva
        36,              // El número total de vértices a dibujar
        GL_UNSIGNED_INT, // El tipo de los indices
        0                // Offset a la hora de leer los indices
    );

    glfwPollEvents();
    glfwSwapBuffers(window);
}
```

¡Ejecuta el programa para darle vida a tu primer cubo!

A ver, yo sé que a los hijos hay que quererlos, pero nos ha salido feillo. Hasta parece que sea 2D. Vamos a hacer que se gire un poco para apreciar su tridimensionalidad. Dentro del bucle, antes de mandar la matriz modelo en el uniform, vamos a rotarla con glm::rotate.

```
// Rotamos el cubo 45 grados en el eje X
model = glm::rotate(model, glm::radians(45.0f),
      glm::vec3(1.0f, 0.0f, 0.0f));
```

Si ejecutas ahora verás que... ¿Que el cubo parpadea? ¿Y eso? Bueno, lo que está pasando es que está rotando a gran velocidad de forma constante. Ten en cuenta que cada iteración le estás aplicando la rotación. Prueba a, en vez de 45 grados, que rote a 1 solo grado para que puedas apreciar mejor este efecto.

Sin embargo, nosotros queríamos que se rote, pero no que se quede girando para siempre. ¿Cómo hago que rote sólo una vez? Muy fácil, vamos a resetear la matriz modelo al principio de cada iteración del bucle.

```
// Reseteamos la matriz en el bucle antes de rotarla
model = glm::mat4(1.0f);
```

Ahora ya tienes a tu primer cubo rotado, ¡y es todo tuyo!

Antes de avanzar te recomiendo que juegues un poco con los ejes de rotación y los ángulos, también prueba a ver qué pasa si estas transformaciones se las haces a la matriz vista (osea, en vez de mover el cubo, mover tu cámara). De esta manera, interiorizarás mejor estos conceptos.

¿Que si ya hemos acabado? Bueno, te prometí que pondríamos un par de cubos más en la escena y lo prometido es deuda. La idea va a ser poner 10 cubos de distintos colores y hacerlos rotar a velocidades distintas velocidades. Con todo lo que hemos aprendido va a ser pan chupado ya lo verás.

Si vamos a tener 10 cubos, necesitaremos 10 posiciones (puedes hacerlas aleatorias si eres un alma caótica pero así nos aseguramos que los 10 cubos se vean en la escena).

```
// Guardamos las posiciones de todas las cajas que vamos a iterar
glm::vec3 cubePositions[] = {
    glm::vec3(0.0f, 0.0f, 0.0f),
    glm::vec3(2.0f, 5.0f, -15.0f),
    glm::vec3(-1.5f, -2.2f, -2.5f),
    glm::vec3(-3.8f, -2.0f, -12.3f),
    glm::vec3(2.4f, -0.4f, -3.5f),
    glm::vec3(-1.7f, 3.0f, -7.5f),
    glm::vec3(1.3f, -2.0f, -2.5f),
    glm::vec3(1.5f, 2.0f, -2.5f),
    glm::vec3(1.5f, 0.2f, -1.5f),
    glm::vec3(-1.3f, 1.0f, -1.5f)
};
```

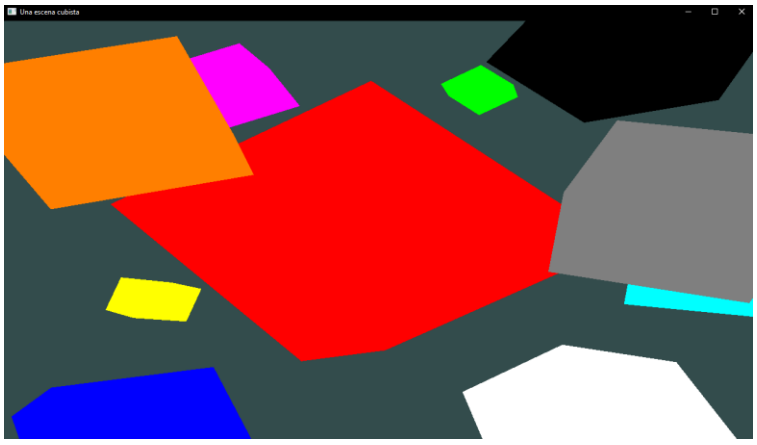
También necesitaremos 10 colores, claro. Buen momento para usar nuestro tipo casero, el que creamos al principio.

```
// Guardamos los colores de cada cubo
Color cubeColors[] = {
    Color(1.0f, 0.0f, 0.0f, 1.0f), // Rojo
    Color(0.0f, 1.0f, 0.0f, 1.0f), // Verde
    Color(0.0f, 0.0f, 1.0f, 1.0f), // Azul
    Color(1.0f, 1.0f, 0.0f, 1.0f), // Amarillo
    Color(0.0f, 1.0f, 1.0f, 1.0f), // Cian
    Color(1.0f, 0.0f, 1.0f, 1.0f), // Magenta
    Color(1.0f, 1.0f, 1.0f, 1.0f), // Blanco
    Color(0.0f, 0.0f, 0.0f, 1.0f), // Negro
    Color(0.5f, 0.5f, 0.5f, 1.0f), // Gris
    Color(1.0f, 0.5f, 0.0f, 1.0f)  // Naranja
};
```

Ahora es tan fácil como dentro, del bucle de renderizado, ir iterando en el array de posiciones, trasladando cada cubo a su propia posición, rotándolo y finalmente mandando un color distinto en le uniform cada vez.

Para la rotación, usaré glfwGetTime que devuelve el tiempo de ejecución del programa, así cada uno de los cubos rotará a velocidades distintas porque, aunque sea por milésimas de segundo, habrá diferencia (pongo una comprobación para que el primero no multiplique por i porque si no en la primera iteración el cubo se quedaría quieto al multiplicar por 0).

```
// Le indicamos a OpenGL que vamos a usar nuestro shader
ourShader.use();

// Iteramos una vez por cada caja
for (unsigned int i = 0; i < 10; i++) {
    // Inicializamos la matriz modelo, para que vuelva a 0 cada vez
    glm::mat4 model{ glm::mat4(1.0f) };
    // Cogemos las posiciones de nuestro array de posiciones
    // según el bucle y lo usamos para trasladar la matriz modelo
    model = translate(model, cubePositions[i]);
    // Un Angulo de rotación diferente para cada cubo y según
    // el tiempo de ejecución
    float angle{};
    if (i > 0) {
        angle = glfwGetTime() * 20.0f * i;
    }
    else {
        angle = glfwGetTime() * 20.0f;
    }
    // Lo rotamos según ese angulo para que este girando
    model = glm::rotate(model, glm::radians(angle),
      glm::vec3(1.0f, 0.3f, 0.5f));
    // Metemos las matrices en los uniforms
    ourShader.setMatrix4fv("model", value_ptr(model));
    ourShader.setMatrix4fv("view", value_ptr(view));
    ourShader.setMatrix4fv("projection", value_ptr(projection));
    // Metemos un color distinto cada vez, siguiendo el bucle
    ourShader.set4Float("ourColor", cubeColors[i].r, cubeColors[i].g,
      cubeColors[i].b, cubeColors[i].a);

    // Dibujamos cada cubo
    glDrawElements(
        GL_TRIANGLES,     // El tipo de primitiva
        36,               // El número total de vértices a dibujar
        GL_UNSIGNED_INT, // El tipo de los indices
        0                 // Offset a la hora de leer los indices
    );
}
```



Parece que por fin todo está bien... Bueno, si has sido un alumno bueno y te has tomado el tiempo de entender las posiciones donde hemos colocado cada cubo en el espacio, te habrás dado cuenta de que el cubo rojo a pesar de que debería ser el más cercano a nosotros se pinta como el más alejado. Eso es que aún nos queda una última aclaración rápida, el testeo de profundidad.

3.5.3. El Z-Buffer

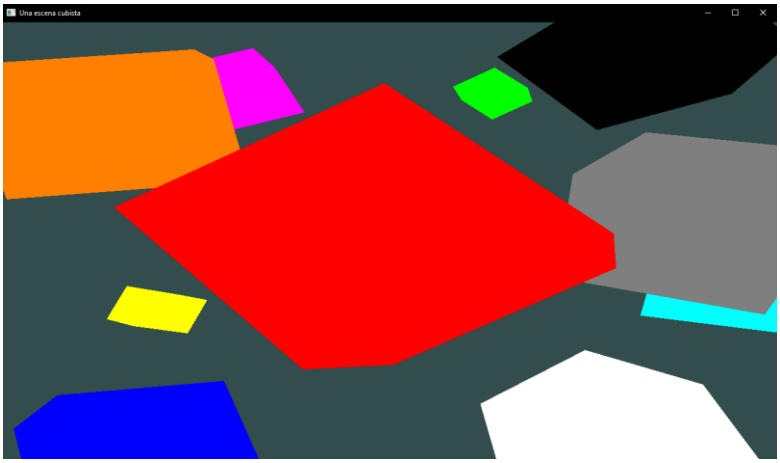
Si lo activas, OpenGL automáticamente guarda la información de la profundidad relativa de un fragmento en un buffer del mismo tamaño que el del color, permitiendo, en un rango de [0,1], saber qué tan lejos está de la cámara el fragmento. Este es conocido como el Z-Buffer.

OpenGL puede luego comparar los valores para saber qué fragmento va delante y cuál detrás (si no, los pintará en el orden que los hayas mandado, que igual para algoritmos como el del pintor es justo lo que quieres). Esta función viene desactivada por defecto. Para activarlo es muy simple.

```
// Activamos el testeo de profundidad antes del
//   bucle de renderizado
glEnable(GL_DEPTH_TEST);
```

Lo único que tenemos que tener en cuenta es que entonces a la hora de pintar no sólo lo estamos haciendo en el buffer de color, sino también el Z-Buffer. Por lo que en cada iteración tenemos que limpiarlo. Con esto, los cubos aparecen como deberían.

```
// Ahora tenemos que limpiar tambien el Z-Buffer
//   ahora cada iteracion
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```



¡Esto sí es una escena cubista en honor al título del capítulo! ¿Ves como no era tan dificil? Es broma, este capítulo ha sido un salto interesante de dificultad, así que siéntete orgulloso si has llegado hasta aquí. Como antes, te dejo el link al código completo por aquí.[13]

[13] Código fuente Fase 2

4. HORA DE ORGANIZAR

Eso ha sido intenso, ¿eh? Hemos aprendido de un plumazo a dibujar primitivas. Como ya dije, esto no es un libro de OpenGL, me voy a tomar la libertad de seguir avanzando en lo importante para un motor como tal. Sin embargo, te recomiendo coger el código del capítulo anterior y jugar con él, cambiar los cubos, dibujar triángulos en vez de cubos, puntos, líneas...

Pronto te darás cuenta que es todo un pequeño caos. Arrays enormes con vértices e índices, los buffers configurados a mano cada vez y que es poco escalable (imagínate gestionar así modelos complejos, texturas, luces...). Esto, para ti que eres muy listo y guapo, igual no es problema, pero piensa en tu pobre usuario que no tiene por qué tener idea de siquiera que estás usando OpenGL.

Como motor, se lo tenemos que poner fácil al usuario. Para eso, dejaremos de pensar en "cubos" y pensaremos en "entidades". Con esto en mente, necesitaremos dos cosas: Guardarlas y organizarlas. En este capítulo reharemos la escena del capítulo anterior pero mejorada con esta filosofía.

4.1. EL RESOURCE MANAGER, TU SECREATARIO

El manejador de recursos es un patrón de diseño muy común en aplicaciones como videojuegos. Cada una de nuestras entidades va a ser un recurso a almacenar: Un shader, un cubo, un modelo complejo... Piensa en el resource manager como tu propio secretario al que le pides "Ey, tráeme el cubo Juan", va al archivo a buscar ese cubo y, si existe, te lo devuelve. Si no, lo crea nuevo.

Esto permite que, como tal, en tu VRAM sólo cargas una vez cada entidad y luego accedes a ese espacio de memoria cada vez que la intentes recuperar. Además, como para comparar usaremos un nombre, podemos darle la oportunidad al usuario de llamar a las entidades como quiera, lo que es un buen caramelito.

Para empezar, pensemos en qué recursos necesitamos para la escena anterior: Un recurso shader y un recurso modelo (el cubo).

4.1.1. Recurso Genérico

Te mentí, antes de empezar lo que realmente vamos a hacer es definir el recurso genérico. Este va a ser una clase de la que luego van a heredar todos los demás recursos, por lo que definirá sus comportamientos comunes. De nuevo, te volví a mentir, a mí me gusta antes de todo esto hacer una clase AllResources.

Esta clase tendrá un #include de todos los recursos conocidos, empezando por el genérico. De esta forma tienes todos los recursos en un mismo header, permitiéndote siempre incluirlos todos de golpe y tenerlos localizados en un mismo punto, facilitando su expansión en el futuro.

```
#pragma once

// Un enumerado con los tipos actuales posibles de recursos,
// más concretos que simplemente "Modelo". Para el futuro.
enum RType{
    RShader, RCube
};

#include "R_Resource.hpp"
#include "R_Shader.hpp"
#include "R_Model.hpp"
```

Por si no lo sabías, el include es a efectos prácticos un copia-pega. Es decir, cuando le dices "#include clase.hpp" le estás diciendo "cópiame todo clase.hpp y pégamelo aquí". Por eso, cuando incluyamos este AllResources será como copiar-pegar todos tus recursos de golpe.

Ahora sí, vamos a trabajar en ese recurso genérico. Como has visto en el AllResoruces, los estoy llamando R_ y el nombre del recurso (Por lo tanto, he llamado al AllResources como R_AllResources). Tú maneja los nombres como quieras, de momento, como si quieres dejar placeholders.

Esta clase tendrá que tener los métodos generales, sea la carga y la asignación y recuperación del nombre del recurso:

```cpp
#pragma once

// Metodo general de recursos, que lo heredaran todos los recursos
struct R_Resource{
    // Destructor por defecto, importante para la gestión
    // de los smartpointers (luego te explico)
    virtual ~R_Resource() = default;
    // Recupera el nombre del recurso
    std::string getName(){return name;};
    // Setea el nombre del recurso
    void setName(std::string name_){name = name_;};
    // Método de carga genérico, los parámetros tendrán más o
    // menos valor según el recurso. El "= 0" obliga a todas las
    // las clases que hereden del genérico a implementar su
    // propia versión de este método
    virtual void load(const std::string& name, RType rt) = 0;

    // Nombre del recurso, importante para no repetir recursos
    std::string name{};
};
```

4.1.2. Recurso Shader

Este recurso es el más sencillo, en el capítulo anterior prácticamente ya hicimos todo el trabajo. El constructor clase Shader que creamos en el apartado 3.5.1. devuelve un objeto Shader que podemos almacenar como nuestro recurso creado, por lo que la clase R_Shader se apoya en esta para sus métodos. Lo único que hay que tener en cuenta es que hereda de nuestro recurso genérico.

```cpp
#pragma once
#include "../Shaders/Shader.hpp"

// Clase de recruso Shader
struct R_Shader: virtual public R_Resource{
    // Constructor por defecto
    R_Shader(){};
    // Constructor con la ruta a un archivo de un vertex
    // y de un fragment shader
    R_Shader(const char* vertex, const char* fragment){
        // Recumeramos el constructor de la clase Shader
        shader = Shader{vertex, fragment};
    }

    // Cargamos igual que en el constructor, con ayuda
    // del constructor de la clase Shader
    void load(const std::string& path, RType rt){
        shader = Shader(path.c_str(), path.c_str());
        // Siempre será RShader pero es sano y escalable mantenerlo
        type = rt;
    };

    // El shader recuperado, almacenado aqui
    Shader shader{};
    // Tipo del recurso
    RType type{};
};
```

4.1.3. Recurso Malla

Efectivamente, en este capítulo estoy un poco mentiroso de más. ¿Recurso malla, no era recurso modelo? Bueno, recuerda que un modelo complejo está hecho de un número indeterminado de mallas. Estas también las vamos a guardar. Piensa en el modelo de una botella, puede ser toda una malla o venir el plástico separado del tapón. Esta será heredada después en el recurso modelo.

Lo primero que vamos a hacer es una pequeña clase Vertex con todo lo necesario para representar nuestra malla. Ahora mismo sólo es un punto 3D en el espacio, pero más adelante será útil cuando usemos más información.

```
// Clase Vertex, que contiene la informacion de un punto
// 3D necesario para el renderizado en OpenGL
struct Vertex{
    glm::vec3 position{};
};
```

Ahora ya podemos empezar con nuestra clase malla. Lo primero, necesitaremos un vector en el que guardar nuestros vértices y nuestros índices. También almacenaremos unos unisgned int para el VAO, el EBO y el VBO. Una vez los tengamos, creamos un método que configure los buffers como aprendimos en el capítulo anterior.

```
// Los vectores con la informacion almacenada
std::vector<Vertex> vertices{};
std::vector<unsigned int> indices{};

private:
    // Seteamos los buffers
    void setupMesh(){
        glGenVertexArrays(1, &VAO);
        glGenBuffers(1, &VBO);
        glGenBuffers(1, &EBO);

        glBindVertexArray(VAO);
        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        glBufferData(GL_ARRAY_BUFFER, vertices.size() *
           sizeof(Vertex), &vertices[0], GL_STATIC_DRAW);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() *
           sizeof(unsigned int), &indices[0], GL_STATIC_DRAW);

        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
           sizeof(Vertex), (void*)0);

        glBindVertexArray(0);
    };

    // Guardamos los indices de los buffers que hemos creado
    unsigned int VAO{}, VBO{}, EBO{};
```

Al usar la longitud y el tamaño de los vectores, y apoyarnos en nuestro tipo Vertex, volvemos la creación de los buffers dinámica. Así no necesitamos meterle nosotros a mano los valores como en el capítulo anterior.

Ahora le añadimos un método público de dibujado que, de nuevo, se basa en lo que ya sabemos del dibujado de triángulos.

```
// Metodo de dibujado en OpenGL. Pasamos el shader
// porque en el futuro lo necesitaremos por aquí
void draw(Shader& shader){
    // Anidamos el Vertex Array Object
    glBindVertexArray(VAO);
    // Dibujamos los triangulos segun los indices
    glDrawElements(GL_TRIANGLES, static_cast<int>
      (indices.size()), GL_UNSIGNED_INT, 0);
    // Desanidamos el Vertex Aray Object
    glBindVertexArray(0);
};
```

Igual que antes, se vuelve dinámico el dibujado al usar la información de nuestros vectores. Por último, un constructor y un método que resetee los buffers por si el usuario ha decidido toquetear las mallas ya creadas.

```
// Constructor con un vector de Vertex, indices y texturas
R_Mesh(std::vector<Vertex> vertices, std::vector<unsigned
  int> indices){
    this->vertices = vertices;
    this->indices = indices;
    // Cuando tenemos los datos los procesamos
    setupMesh();
};

// Por si el usuario ha cambiado la mesh y quiere volver a
// setearlo todo internamente
void changeMesh(){setupMesh();};
```

Por si hay algo que no te haya quedado del todo claro, te dejo un enlace a la clase[14] para que puedas jugar con ella.

4.1.4. Recurso Modelo

Ahora por fin ya podemos crear nuestro último recurso, el modelo. El cubo. Como atributo, podemos decir el tipo de recurso (Con el RType que hicimos antes, aunque de momento sólo existe el modelo cubo), sus mallas y un puntero al shader con el que se va a dibujar (De momento el shader nos es irrelevante, pero ya le daremos uso en el capítulo que viene).

[14] Archivo R_Mesh.hpp

```
// Tipo del recurso
RType type{};
// Vector de mallas
std::vector<R_Mesh> meshes{};
// Puntero al Shader
Shader* shader{};
```

Definiremos el método de carga. De momento vamos a imaginarnos que no sólo vamos a cargar un cubo sino un modelo más complejo, del cual nos habrán mandado la ruta del archivo.

```
// Metodo de cargado, con la ruta del archivo y
// el tipo de archivo
void load(const std::string& path, RType rt) override{
    // Asignamos el tipo de recurso
    type = rt;
    // Segun el tipo de recursos hace una cosa u otra
    if (rt == RType::RCube) {
        // Metemos en una mesh el cubo
        meshes.push_back(processCube());
    }
};
```

Como ves, tratamos el modelo según el tipo. De momento sólo existe el cubo, así que le vamos a añadir una única malla (un cubo entero). En ese método “processCube” tenemos lo que hicimos el capítulo anterior con los vértices y los índices.

```
// Procesa el cubo
R_Mesh processCube(){
    std::vector<Vertex> vertices{};
    std::vector<unsigned int> indices{};

    // Cara delantera (z = 0.5)
    // Vértice 0 (Esquina superior izquierda)
    vertices.push_back(Vertex{glm::vec3(-0.5f, 0.5f, 0.5f)});
    // Vértice 1 (Esquina superior derecha)
    vertices.push_back(Vertex{glm::vec3( 0.5f, 0.5f, 0.5f)});
    // Vértice 2 (Esquina inferior izquierda)
    vertices.push_back(Vertex{glm::vec3(-0.5f,-0.5f, 0.5f)});
    // Vértice 3 (Esquina inferior derecha)
    vertices.push_back(Vertex{glm::vec3( 0.5f,-0.5f, 0.5f)});

    // Cara trasera (Eje Z = -0.5)
    // Vértice 4
    vertices.push_back(Vertex{glm::vec3(-0.5f, 0.5f,-0.5f)});
    // Vértice 5
    vertices.push_back(Vertex{glm::vec3( 0.5f, 0.5f,-0.5f)});
    // Vértice 6
    vertices.push_back(Vertex{glm::vec3(-0.5f,-0.5f,-0.5f)});
    // Vértice 7
    vertices.push_back(Vertex{glm::vec3( 0.5f,-0.5f,-0.5f)});

    ...
```

```
    ...

    // Cara delantera
    indices.push_back(3);
    indices.push_back(2);
    indices.push_back(0);
    indices.push_back(3);
    indices.push_back(0);
    indices.push_back(1);

    // Cara trasera
    indices.push_back(7);
    indices.push_back(6);
    indices.push_back(4);
    indices.push_back(7);
    indices.push_back(4);
    indices.push_back(5);

    // Cara izquierda
    indices.push_back(7);
    indices.push_back(5);
    indices.push_back(1);
    indices.push_back(7);
    indices.push_back(1);
    indices.push_back(3);

    // Cara derecha
    indices.push_back(6);
    indices.push_back(4);
    indices.push_back(0);
    indices.push_back(6);
    indices.push_back(0);
    indices.push_back(2);

    // Cara inferior
    indices.push_back(7);
    indices.push_back(7);
    indices.push_back(2);
    indices.push_back(7);
    indices.push_back(2);
    indices.push_back(6);

    // Cara superior
    indices.push_back(5);
    indices.push_back(4);
    indices.push_back(0);
    indices.push_back(5);
    indices.push_back(0);
    indices.push_back(1);

    // Devuelve la mesh
    return R_Mesh(vertices, indices);
}
```

Igual el método asusta de primeras, pero si lo miras con detenimiento es muy sencillo. A la clase sólo le queda un método de dibujado que invoque a cada una de sus mallas para que ellas solas se dibujen, que si recuerdas antes ya le hicimos el método de dibujado.

```
// Metodo de dibujado, requiere de un shader para el
// futuro, de momento no es útil
void drawWithShader(Shader& shader){
    // Para cada una de sus mallas, llama a su método
    // de dibujado con el mismo shader
    for(unsigned int i{}; i<meshes.size(); i++)
        meshes[i].draw(shader);
};
```

Con esto ya tenemos el recurso modelo. Por si te has perdido con algo, de nuevo, te dejo por aquí[15] la clase.

Ya tenemos todos los recursos, por lo que podremos hacer nuestro gestor al fin. Pero antes, un pequeño espacio para nuestro patrocinador: Los punteros inteligentes.

4.1.5. Punteros Inteligentes

C++ moderno a veces da miedo, pero puede ser muy útil en ciertas ocasiones. ¿Recuerdas lo molesto que es estar haciendo new y delete para manejar la memoria de los punteros? ¡Pues ya no más, los punteros inteligentes se encargan de hacer ese trabajo por ti! El tema es un poco más complejo, pero de momento me sirve con que te quedes con los dos principales:

- Unique_Ptr: El puntero posee de forma exclusiva al objeto al que apunta, es decir, nadie más puede copiarlo. Cuando se destruya el puntero, se destruirá el objeto con él.
- Shared_Ptr: El puntero comparte al objeto que apunta. El propio puntero almacena dinámicamente con cuántos lo ha compartido. Sólo se destruirá el objeto asociado cuando no quede ningún puntero de los que le apuntaban a él vivo.

Es importante entender todo esto ahora porque los vamos a usar bastante.

4.1.6. El Gestor de Recursos

El concepto es sencillo: El usuario te pide un recurso, lo buscas entre tus recursos guardados. ¿Está? Devuelves una referencia al recurso. ¿No está? Lo cargas como uno nuevo y devuelves una referencia tras cargarlo. Vamos a empezar por la cabecera que es lo complicado y luego tiraremos por la implementación.

[15] Archivo R_Model.hpp

Lo primero, nuestro vector de recursos, los guardaremos genéricos para que dé igual el tipo. Generalmente usaríamos Unique_Ptr para que no pudiesen hacer copias del objeto y arriesgarnos a que se quede en memoria si no se han borrado todos los punteros, pero más adelante te explicaré por qué he usado Shared_Ptr.

```
private:
    std::vector<std::shared_ptr<R_Resource>> resources;
```

En cuanto al método de recuperar los recursos, puedes tomar dos rutas distintas: Hacer una planitlla genérica para todos los recursos o hacer una implementación del método para cada tipo de recurso diferente. Como comprenderás, la primera es la mejor. Aunque para felicitarnos la vida (que ya te la he complicado bastante) voy a separar el recurso shader en un método diferente, por claridad.

```
// Metodos de carga, son plantillos que luego se aprovecharan
// más cuando separemos los modelos en tipos más específicos
template <class ResourceType>
ResourceType* getResource(std::string name, std::string src,
  RType rt){
    // Buscamos en el vector de recursos con un iterador
    std::vector<std::shared_ptr<R_Resource>>::iterator it =
      std::find_if(resources.begin(), resources.end(),
      [&] (std::shared_ptr<R_Resource> r){
        // Comprobamos por el nombre
        if(r->name == name){
            return true;
        }
        return false;
    });
    // Si ha encontrado, no entra en este if, si no ha
    // encontrado ninguno con ese nombre, lo crea
    if(it == resources.end()){
        // Crea el nuevo recurso como un Shared_Ptr
        auto newRec = std::make_shared<ResourceType>();
        // Llama al cargador de la clase
        newRec->load(src, rt);
        // Le asigna el nombre
        newRec->setName(name);
        // Actualiza el vector de recursos, pero lo guarda
        // casteándolo del tipo específico al tipo genérico
        // con Static_Pointer_Cast
        resources.push_back(
          std::static_pointer_cast<R_Resource>(newRec));
        return newRec.get();
    }
    // Si lo ha encontrado lo devuelve, casteándolo del tipo
    // genérico al específico con Dynamic_Pointer_Cast
    return  std::dynamic_pointer_cast<ResourceType>(*it).get();
}
```

Aunque pueda parecer complicada, la dinámica del casteo es sencilla. Como nuestro vector de recursos es de tipo genérico, pero al usuario le queremos dar

su tipo específico (por ejemplo, R_Model), cuando guardamos un nuevo recurso en el vector lo casteamos del tipo específico al genérico (R_Model -> R_Resource) y cuando recuperamos un recurso del vector lo casteamos al tipo específico (R_Resource -> R_Model). Para poder usar estos casteos y aprovechar el polimorfismo, necesitamos que el puntero sea un Shared_Ptr, de ahí que los use en el vector de recursos.

Como he dicho antes, vamos a hacer una versión para el recurso R_Shader, más que nada para ir rápidos y no tener que redefinir constructores. A veces hay excepciones y es mejor tratarlas por separado, así que no te vendrá mal tampoco saber crear una template hermana.

```
template <class ResourceType>
ResourceType* getResource(std::string name, std::string vs,
  std::string fs){
    std::vector<std::shared_ptr<R_Resource>>::iterator it =
      std::find_if(resources.begin(), resources.end(),
      [&] (std::shared_ptr<R_Resource> r){
        if(r->name == name){
            return true;
        }
        return false;
    });
    if(it == resources.end()){
        // Como es sí o sí un R_Shader por los parámetros,
        // sabemos que el constructor recibirá los parámetros
        // como el constructor de la clase Shader de la Fase 2
        auto newRec = std::make_shared<ResourceType>(
          vs.c_str(), fs.c_str());
        newRec->setName(name);
        newRec->setScene(scene);
        resources.push_back(
            std::static_pointer_cast<R_Resource>(newRec));
        return newRec.get();
    }
    return  std::dynamic_pointer_cast<ResourceType>(*it).get();
}
```

Por último, definimos los métodos get que va a usar el usuario. Esos métodos que usará para pedirnos recursos.

```
// Busca si existe un modelo o un shader, si no lo crea
R_Model* getModel(std::string name, std::string src, RType rt);
R_Shader* getShader(std::string name, std::string vs,
      std::string fs);
```

Para estos métodos sí que necesitaremos definir su implementación en un archivo .cpp. En Windows no te preocupes que Visual Studio te lo compilará por ti, en Linux tendrás que asegurarte de añadirlo al make.

```
// Llamada a la funcion de recoger recurso segun un modelo
R_Model* ResourceManager::getModel(std::string name,
  std::string src, RType rt) {
    return getResource<R_Model>(name, src, rt);
}

// Llamada a la funcion de recoger recurso segun un shader
R_Shader* ResourceManager::getShader(std::string name,
  std::string vs, std::string fs) {
    return getResource<R_Shader>(name, vs, fs);
}
```

Y ya está, gestor de recursos terminado. Ya tenemos los distintos recursos que maneja de momento nuestro motor y los métodos para crearlos/recuperarlos. Ahora el usuario ya no tiene que configurar los buffers, los vértices y los índices, nosotros lo hacemos por él y encima de forma organizada, olé. Te dejo el código de la cabecera[16] y la implementación[17] por si te has perdido con algo.

Un detalle antes de terminar, quiero aclarar que por el bien de no complicar la explicación he usado siempre vectores que, aunque intuitivos y útiles, quizá otras estructuras como mapas desordenados sean más óptimas. Te animo a que las uses si quieres que tu motor sea más guay.

4.2. EL SCENE TREE, TU DIRECTOR

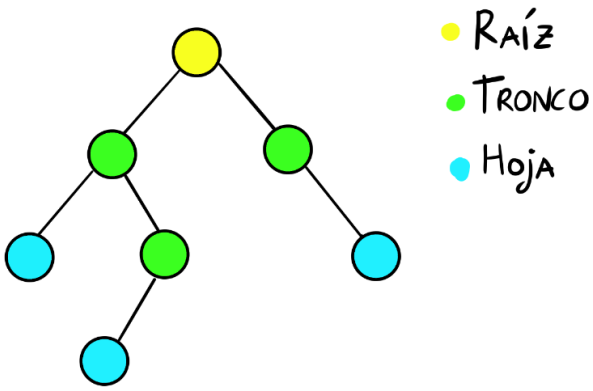
Si el gestor de recursos era tu secretario, el árbol de la escena es tu director. Hemos dicho que teníamos que entender los cubos y los objetos como entidades, no sólo como recursos. Piensa en una entidad como una instancia de un objeto en el mundo. El árbol de la escena se encargará de ordenarlas, organizarlas y, cuando toque pintar por pantalla, se encargará de ir una a una dándole órdenes.

Lo importante del árbol no es sólo el orden, sino también la jerarquía. Permitir nodos padres y nodos hijos ayuda a que, si aplicas una transformación a la familia, se aplica a todos. Si alguna vez has trabajado con un árbol, sabrás que hay distintos tipos de árboles y distintas formas de recorrerlos. No voy a entrar en detalle, porque no es relevante, pero para tu información nuestro árbol de la escena será un N-Árbol y la recorreremos de manera trasversal (primero el padre, y luego los hijos de forma recursiva).

[16] Archivo ResourceManager.hpp
[17] Archivo ResourceManager.cpp

4.2.1. Nodos del árbol

Todo árbol que se precie tiene su raíz, sus ramas y sus hojas. En programación, la raíz la entendemos como el punto principal del que nace el árbol, las ramas son los hijos de la raíz que también tienen hijos y las hojas son las últimas de la cadena, aquellos que ya no tienen hijos.



Independientemente de qué parte del árbol sea, cada circulito (o entidad) la llamaremos nodo. El único nodo especial será la raíz, a la cual la llamaremos raíz de la escena, y será un nodo recipiente para nuestra cámara principal. Hay que tener en cuenta que tenemos muchas cámaras, pero en principio y a no ser que desarrolles un sistema multicámara, sólo podrás ver por una a la vez. Esa cámara elegida, es la cámara principal.

Dicho esto, si pensamos en todo lo que tiene que tener un nodo, podemos pensar que primero tendrá que tener un puntero a su entidad (que aún no hemos llegado a ellas), un vector de hijos si es que tiene, un padre (excepto la raíz de la escena, todos tendrán padre) y un puntero a la cámara principal.

```
// Cada nodo puede tener un tipo de entidad asignada
bool setEntity(E_Entity* newEntity);
E_Entity* getEntity(){return entity;};

// Método para recuperar y asignar una cámara principal,
// útil sólo en la raíz
void setPrincipalCamera(E_Camera* newPCamera){
      principalCamera = newPCamera;};
E_Camera* getPrincipalCamera(){return principalCamera;};

private:
    // Almacenamiento de la estructura del árbol
    // y sus entidades
    E_Entity* entity{};
    std::vector<Node*> childs{};
    Node* dad{};
    E_Camera* principalCamera{};
```

Otra cosa que tendrá que tener el nodo es la capacidad de gestionar sus hijos. Añadir, quitar y obtener. De momento, declaramos los métodos en el header.

```
// Metodo para la gestion de los hijos
uint32_t addChild(Node* newNode);
bool removeChild(Node* remNode);
std::vector<Node*>& getChildren(){return childs;};
Node* getDad(){return dad;};
```

Para la implementación, si ya has trabajado con árboles verás que es sencillo, si es tu primera vez verás que tampoco es mucho misterio. Como el motor es humilde, el árbol de la escena también lo es, lo que facilita mucho el trabajo.

```
// Anade el hijo pasado como parametro al nodo actual y
// devuelve la cantidad de hijos
uint32_t Node::addChild(Node* newNode) {
    // Construye un nuevo nodo al final del vector
    childs.emplace_back(newNode);
    return static_cast<uint32_t>(childs.size());
}

// Elimina del array de hijos el hijo pasado como parametro
// y devuelve si lo ha borrado
bool Node::removeChild(Node* remNode) {
    // Itera para comprobar si el nodo es igual al
    // recibido por parámetro
    auto it = std::find(childs.begin(), childs.end(), remNode);
    if (it != childs.end()) {
        // Si lo es, lo borra y devuelve donde estaba
        childs.erase(it);
        return true;
    }
    return false;
}
```

También vamos a implementar el método setEntity que declaramos antes, también muy sencillo.

```
// Asigna una entidad y devuelve true o false si
// lo ha hecho o no
bool Node::setEntity(E_Entity* newEntity) {
    bool res{};
    if (newEntity != nullptr
      && entity == nullptr) {
        entity = std::move(newEntity);
        res = true;
    }
    return res;
}
```

Con esto ya tenemos las necesidades básicas de nuestro árbol cubiertas. No era tan dificil, ¿eh? Sin embargo, aún no hemos acabado con él. Una cosa importante es cómo vamos a recorrerlo.

Si todos los nodos tienen un método de dibujo y queremos dibujar nuestra escena, simplemente recorriendo el árbol e invocando ese método por cada nodo podemos dibujar todas las entidades de un plumazo. Este método recibirá la cámara desde donde renderizaremos la escena y lo que se conoce como la matriz acumulada, que no es más que su transformación local (lo que conocemos como la matriz modelo) y las de sus ancestros.

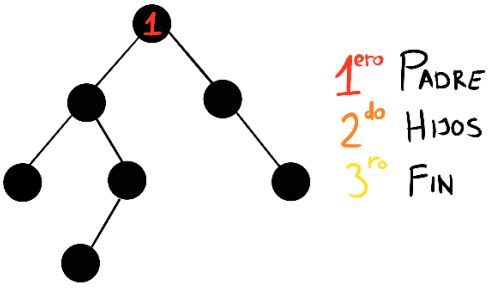
```
// Su transformación local (empieza por la identidad)
glm::mat4 transMatrix{1.0f};


// Metodo que recorre el arbol y dibuja todos sus hijos teniendo
// en cuenta una camara, normalmente de uso exlusivo de la raiz
void traversal(glm::mat4 parentMatrix,
      E_Camera* principalCamera);
```

Sin embargo, ¿cómo dibujamos una cámara? Bueno, como poder puedes, pero como tal una "cámara" es una vista al mundo, no es un objeto. Para diferenciarlas haremos que cada entidad tenga un puntero al gestor de recursos, si este puntero es nulo no se dibuja. Podemos implementar nuestro método transversal con esto en mente.

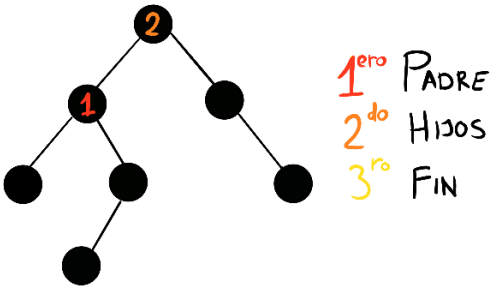
```
// Recorrido del arbol
void Node::traversal(glm::mat4 parentMatrix,
  E_Camera* prCamera) {
    // Le pasa la matriz del padre y recalcula la
    // matriz de transformación, volviéndose la acumulada
    glm::mat4 cumulativeMatrix = parentMatrix * transMatrix;

    // Si tiene entidad y esta entidad tiene un puntero a un
    // ResourceManager que existe, le manda a dibujar.
    if (entity && entity->RM)
        entity->draw(cumulativeMatrix, prCamera);

    // Recursivamente llamamos a los hijos, pasándole la
    // matriz de transformación que se convertirá en la
    // acumulada para su hijo
    for (auto& c : childs)
        c->traversal(cumulativeMatrix, prCamera);
}
```

Como puedes ver, la forma transversal de recorrer el árbol implica recursividad. Es un algoritmo muy simple. Primero pintas el padre, después todos sus hijos. Es recursivo porque cuanto pintes un hijo, se repetirá el algoritmo con él como padre.

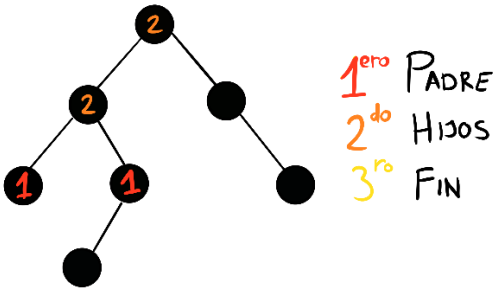
1era Iteración



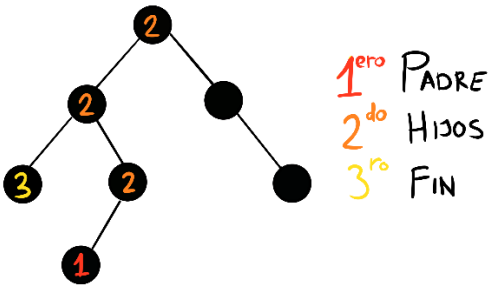
2da Iteración



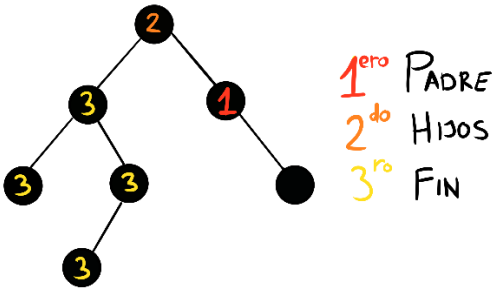
3era Iteración



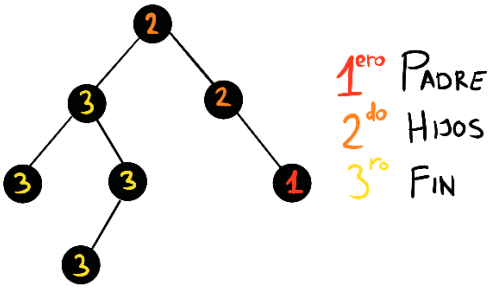
4ta Iteración



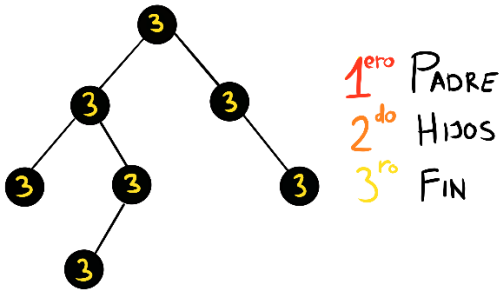
5ta Iteración



6ta Iteración



7ma iteración



Con esto ya podemos recorrer nuestro árbol para dibujar la escena. Solo nos queda tocar un tema, las transformaciones. Como hemos mencionado, cada nodo almacena su matriz de transformación local, esta matriz es la que se corresponde con la modelo. Es decir que, si queremos trasladar, rotar u escalar

una entidad de un nodo, es esta matriz la que tenemos que modificar. Por lo tanto, es la clase nodo la que debe tener definidos todos los métodos de transformación.

```
// Métodos para la transformación del nodo
void translate(glm::vec3 translate);
void rotate(glm::vec4 rotation);
void scale(glm::vec3 scaleM);
// Por si el usuario quiere cambiar o recuperar la matriz
void setTransMatrix(glm::mat4 transMatrix){
      this->transMatrix = transMatrix;};
glm::mat4 getTransMatrix(){return transMatrix;};
```

La implementación también es directa, nada que no hayamos visto ya.

```
// Actualiza la matriz de transformación con la función
// translate de GLM y marca que lo ha actualizado
void Node::translate(glm::vec3 toTranslate) {
    transMatrix = glm::translate(transMatrix,
      glm::vec3(toTranslate.x, toTranslate.y, toTranslate.z));
}

// Actualiza la matriz de transformación con la función
// rotate de GLM y marca que lo ha actualizado
void Node::rotate(glm::vec4 toRotate) {
    // Aquí aprovecho el cuarto elemento del vector para
    // el ángulo, pero si quieres puede separarlo
    transMatrix = glm::rotate(
      transMatrix, glm::radians(toRotate.w),
      glm::vec3{ toRotate.x, toRotate.y, toRotate.z });
}

// Actualiza la matriz de transformación con la función
// scale de GLM y marca que lo ha actualizado
void Node::scale(glm::vec3 toScale) {
    transMatrix = glm::scale(transMatrix,
      glm::vec3(toScale.x, toScale.y, toScale.z));
}
```

Ale, con esto ya tenemos los nodos y la estructura de nuestro árbol. Las transformaciones que nos indican no sólo su posición en la escena sino su posición respecto a sus padres e hijos. Además, el árbol puede dibujar toda la escena al recorrerlo. ¿Ahora me entiendes cuando decía que el árbol era tu director de escena? Te dejo las clases[18] para que no nos perdamos.

Sólo nos quedan tratar las entidades, que hasta ahora simplemente las había mencionado por nombre. Toca profundizar en ellas.

[18] Archivo SceneTree.hpp y Archivo SceneTree.cpp

4.2.2. Entidad Genérica

Igual que con el gestor de recursos, necesitaremos definir nuestras entidades. La filosofía esta vez es qué objetos existirán en la escena, no los que almacenamos en la memoria. Por lo tanto, de momento pensaremos en el modelo (el cubo) y la cámara (nuestra perspectiva). También necesitaremos una clase genérica de entidades para que todas la hereden (el E_Entity de antes) y un E_AllEntidades.

```
#pragma once

// Todos los tipo de entidades

#include "E_Entity.hpp"
#include "E_Camera.hpp"
#include "E_Model.hpp"
```

```
#pragma once

// Forward declaration
struct ResourceManager;
struct E_Camera;

struct E_Entity{
    // Clase de dibujado que van a heredar todas las
    // entidades, obligamos a que la implementen
    virtual void draw(glm::mat4 matrix = glm::mat4(1.0f),
      E_Camera* principalCamera = nullptr) = 0;

    // Puntero al Resource Manager, solo lo tendrán
    // informado las entidades que se relacionen con
    // un recurso (el modelo sí, la cámara no)
    ResourceManager* RM{};
};
```

El forward declaration no es más que decirle a nuestra clase "Mira, confía en mí, existe una clase llamada así pero aún no te voy a decir cómo es". Así podrás usar el nombre de la clase para punteros y referencias antes de definir realmente la clase como tal. De esta forma, puedes definir relaciones entre clases sin necesidad de incluir toda su definición, lo que agiliza la compilación y evita referencias cíclicas.

Ahora nos podemos centrar en las entidades específicas. Estas tienen algo más de chicha que los recursos, sobre todo la cámara, así que empecemos por la entidad modelo que nos sonará un poco más.

4.2.3. Entidad Modelo

Nuestras entidades modelo van a estar directamente relacionadas con nuestros recursos, como es obvio. Antes de crear nuestro modelo en la escena

lo tendremos que cargar. También todo modelo necesitará su shader con el que se va a dibujar, entre otras cosas que nos vendrán bien tener.

```
// Un puntero al modelo y al shader
R_Model* model{};
R_Shader* shader{};
// La relación de aspecto
float aspect{};
// El color por defecto, le he puesto el amarillo
Color color{1.0f, 1.0f, 0.0f, 1.0f};
```

Bien, el constructor será sencillo. Necesitará recibir el gestor de recursos para asignárselo, un nombre para el modelo, la ruta done está almacenado y el tipo. Nuestros cubos los cargamos nosotros a mano en memoria, no se leen de un archivo, pero está bien tenerlo ya implementado para el futuro. A parte le voy a añadir unos cuantos setters.

```
// Constructor del modelo base
E_Model(){};
// Constructor con un gestor de recurso, necesario para el
// dibujado, pero sin un modelo
E_Model(ResourceManager* RM){this->RM = RM;};
// Constructor con un gestor de recurso y un recurso asociado,
// además de un tipo de recurso. La ruta igual al nombre
E_Model(ResourceManager* RM, std::string name, RType type){
      this->RM = RM; model = RM->getModel(name, name, type);};
// Constructor con un gestor de recurso y un recurso asociado,
// ademas de un tipo de recurso. La ruta diferente al nombre
E_Model(ResourceManager* RM, std::string name, std::string src,
      RType type) { this->RM = RM;
      model = RM->getModel(name, src, type); };

// Busca el shader dentro del gesto de recursos
void setShader(std::string name){
      shader = RM->getShader(name, name, name);};
// Setea la relacion de aspecto
void setAspect(float asp){aspect = asp;};
// Asignamos el modelo, el nombre igual a la ruta
void setModel(std::string name, RType type) {
      model = RM->getModel(name, name, type); };
// Asignamos el modelo, el nombre y la ruta diferentes
void setModel(std::string name, std::string src, RType type) {
      model = RM->getModel(name, src, type); };
```

Como ves, me gusta crear distintas variaciones de constructores y setters. Ten en cuenta que los recursos los manejamos internamente nosotros, pero el usuario creará y manipulará entidades. Como motor, cuantas más opciones le demos al usuario, mejor.

Ahora queda el método de dibujado. Aquí rescataremos todo eso de los uniforms y las matrices modelo, vista y proyección. Aunque recordemos que la modelo la recibimos de la matriz acumulada del nodo.

```
// Metodo de dibujado, comunes para todos los modelos
void draw(glm::mat4 matrix = glm::mat4(1.0f),
  E_Camera* principalCamera = nullptr) override{
    // Si hay un shader, un modelo y una cámara principal se dibuja,
    // si no no se podrá
    if(shader && model && principalCamera){
        // Le asigna el shader al recurso modelo
        model->setShader(&shader->shader);
        // Creo una referencia a ese shader, para que sea más fácil de usar
        Shader& actualShader = shader->shader;
        // Activo el shader
        actualShader.use();
        // La mtriz modelo es la matriz acumulaba recibida
        glm::mat4 modelMatrix = matrix;
        // La matriz vista usando el método de la cámara, ya que
        // es a partir de ella desde donde vamos a visualizar la escena
        glm::mat4 viewMatrix = principalCamera->getViewMatrix();
        // Para simplificar, la matriz proyección va a ser siempre
        // perspectiva (Puedes hacer que la informe el usuario si quieres).
        // Usamos el fov de la camara (zoom), la relación de aspecto que ha
        // marcado el usuario y un z_near y z_far que puedes hacer
        // también que sea modular si quieres
        glm::mat4 projectionMatrix = glm::perspective(
            glm::radians(principalCamera->zoom), aspect, 0.1f, 200.0f);
        // Calculamos la modelViewProjection en CPU
        glm::mat4 mvp = projectionMatrix * viewMatrix * modelMatrix;
        // Informamos los uniforms de los shaders
        actualShader.setMat4("modelViewProjection", mvp);
        // Si el usuario ha modificado el color saldrá con el nuevo,
        // si no el de por defecto
        actualShader.setVec4("ourColor",
            color.r, color.g, color.b, color.a);
        // Ahora que hemos seteado el shader llamamos
        // al renderizado del recurso modelo
        model->drawWithShader(actualShader);
    }
};
```

Puede que la cantidad de texto intimide, pero si vas paso a paso verás que realmente es bastante sencillo. Sólo hay una cosa a puntualizar: Los sahders son iguales que en el capítulo anterior, pero esta vez calculamos el producto de las matrices en la CPU (es mejor calcularlo una vez aquí que una vez por vértice en el Vertex Shader), por lo que habrá que modificar el VS para que en reciba la matriz completa calculada en vez de las tres separadas en diferentes uniforms. Como ves, la cámara se usa mucho. Efectivamente, me he dejado lo mejor para el final.

4.2.4. Entidad Cámara

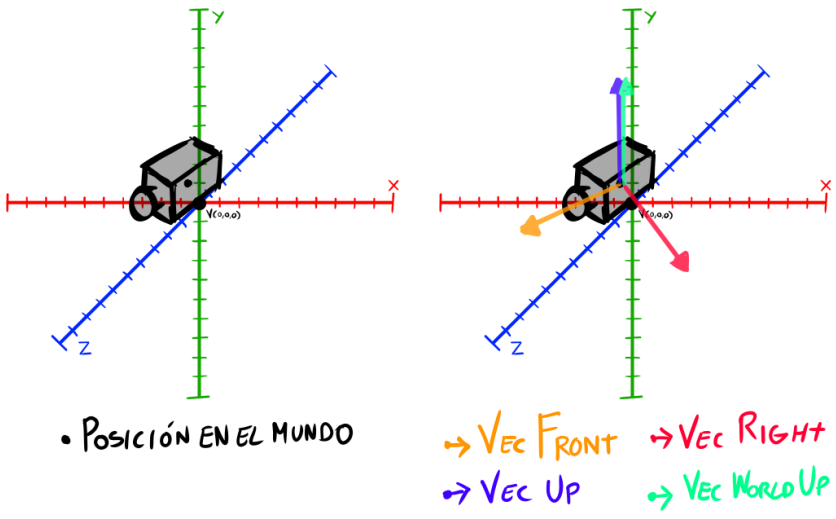
Sin duda, esta es la más compleja. No por nada, sino porque introduce un montón de conceptos nuevos que hasta ahora no habíamos tratado. Hasta ahora simplemente colocábamos los cubos en el mundo y los veíamos desde un punto fijo e inamovible. Es hora de cambiar eso.

Tratar a la cámara como una entidad y no como una simple matriz trae una cantidad de ventajas interesantes, permitiendo un movimiento fluido, anclarlo a un nodo para que "lo siga", tener distintas vistas e ir saltando de una a otra... En definitiva, la cámara es una de las entidades primordiales para un motor gráfico. De hecho, ya has visto que la hemos tenido en cuenta en varios sitios.

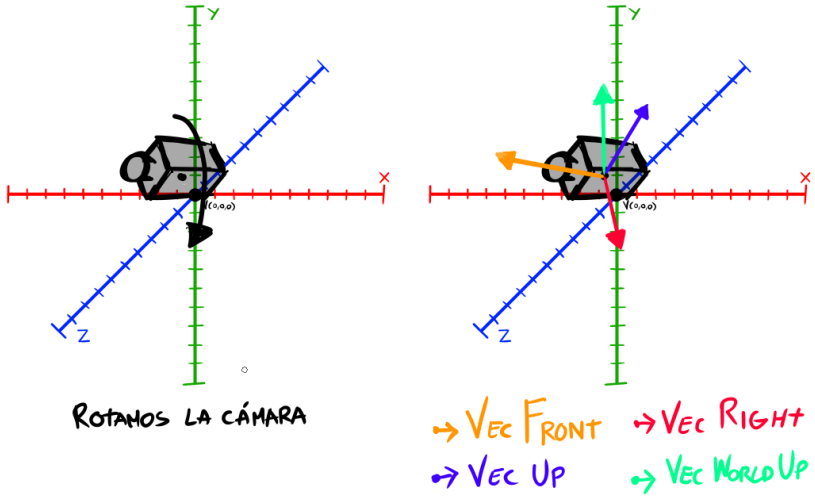
Hay distintos tipos de cámara (primera persona, tercera, isométrica...). De momento, vamos a centrarnos en una cámara de primera persona, el resto sólo necesitará ajustes menores. Para ella, necesitaremos conocer varios atributos: Su posición y varios vectores unitarios de dirección.

```
// La posición en el mundo
glm::vec3 position{};
// Vector que apunta hacia donde mira la cámara
glm::vec3 front{};
// Vector que apunta hacia arriba desde la cámara
glm::vec3 up{};
// Vector que apunta hacia la derecha desde la cámara
glm::vec3 right{};
// Vector que apunta hacia arriba desde la cámara
// respecto al mundo
glm::vec3 worldUp{};
```

Estos vectores son cruciales y representan el estado de la cámara en el mundo. Es importante separar el vector worldUp (el "arriba absoluto" de la escena) del vector up (el "arriba relativo" de la cámara). Imagina una cámara:



Como ves, el vector Up y el WorldUp son idénticos, ambos coinciden con el eje vertical Y positivo. Pero es importante separarlos porque ahora, imagina que rotas la cámara:



De repente, ambos vectores representan información distinta. Por eso lo correcto es separarlos. Ahora te preguntarás, ¿para qué necesito toda esta información? Manteniendo actualizados estos vectores conforme la cámara se vaya desplazando por el mundo, luego podremos hacer una matriz vista que nos permita ver desde la perspectiva de dicha cámara. Pero no nos adelantemos, antes vamos a hablar de esa rotación.

4.2.4.1. Rotación de Euler

Existen tres ángulos de rotación en nuestro sistema. La rotación Yaw, en el eje Y (es como mirar a izquierda/derecha), la rotación Pitch en el eje X (que es como mirar arriba/abajo) y la rotación Roll en el eje Z (que es como ladear la cabeza a los lados). Si has jugado a cualquier First Person Shooter estarás más que familiarizado. Estos ángulos serán los que nos permitan calcular después el vector front, el que indica dónde apunta la cámara. ¿Cómo? Con el poder oculto de las esferas y la trigonometría.

Sin embargo, cuentan las leyendas que Euler tenía un enemigo mortal. Un temible señor oscuro llamado... Gimbal Lock. Este bloqueo no es más que el resultado de que dos de nuestros tres ejes se alineen cuando el tercero alcanza un ángulo de 90 grados, perdiendo la capacidad de distinguirlos. Esto ocurre porque, al estar alineados, si rotas en un eje en la práctica estás rotando para los dos, lo que, aunque parezca trivial, generará mucho comportamiento indefinido.

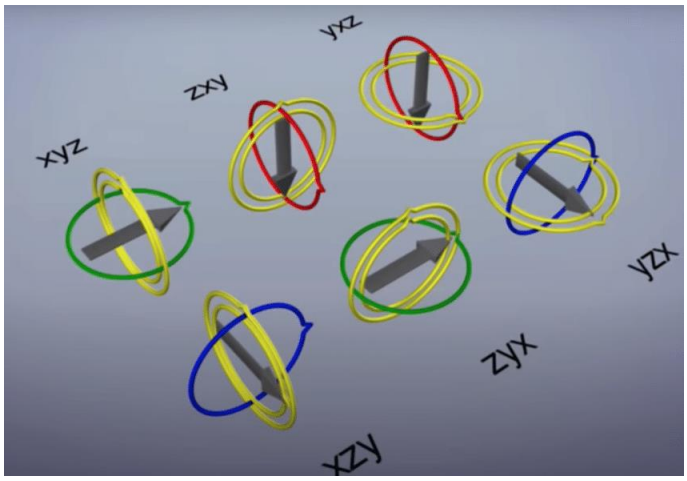


Figura 2: Todos los tipos de Gimbal Lock[19]

Es de estas cosas que son más fáciles de entender si se ven en movimiento, así que, si no te ha quedado claro con mi explicación y la foto, te recomiendo que te mires el video de donde la he sacado[19].

¿Cómo nos libramos de Gimbal Lock? Bueno, no podemos. Es un riesgo intrínseco de los ángulos de Euler. Sin embargo, podemos circunnavegarlo. Por ejemplo, lo primero que vamos a hacer es olvidarnos del Roll. En una cámara en primera persona, casi nunca se usa esta rotación. Usando sólo dos de los ángulos podemos evitar el Gimbal Lock si evitamos que cualquiera de esos dos llegue a 90 grados. Para nuestra primera cámara, vas más que sobrao así.

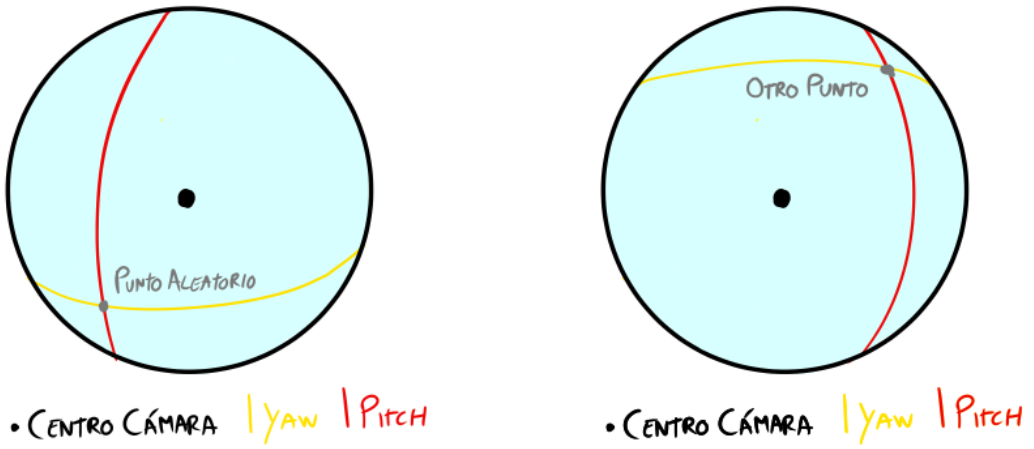
Si quieres de verdad evitar el Gimbal Lock y que tu cámara cual simulador espacial se mueva libre en todos los ejes, vas a tener que adentrarte en el maravilloso mundo de los cuaterniones. Suenan más intimidante de lo que es, pero es cierto que añade una capa de complejidad. Te dejo alguna referencia por si te animas[20].

4.2.4.2. Vector Front usando Euler

Vale, hemos vencido al Gimbal ese, pero todavía no te he dicho cómo sacamos el vector front de los ángulos de Euler. Bueno, estaba intentando protegerte de las matemáticas, pero ya no vas a poder escapar. Vamos a desengranar parte por parte cómo sacar el vector front necesario para nuestra matriz vista.

[19] Fuente: Euler (gimbal lock) Explained
[20] Camera Rotation With Quaternions

Imagínate una esfera de radio 1 alrededor de nuestra cámara. Cualquier punto de esta esfera se puede definir por la intersección de dos ángulos, casualmente el Yaw y el Pitch.



Para calcular la altura, es tan fácil como conocer el seno del Pitch, así cuando Pitch sea 0 la altura también lo será y cuando sea ±90º la altura será ±1, el otro extremo.

```
frontVec.y = static_cast<float>(sin(glm::radians(pitch)));
```

El problema son los ejes X y Z. Tendremos que proyectar las coordenadas esféricas a sus planos cartesianos. Empezamos por el X.

```
frontVec.x = static_cast<float>(
      cos(glm::radians(yaw)) *
      cos(glm::radians(pitch))
);
```

Aquí pivotamos sobre el Pitch. Cuando el Pitch sea 0 (no hay angulación vertical, miras al frente) su coseno será 1 y por lo tanto todo el valor de X dependerá del Yaw. Por su contrario, cuando el Pitch sea de ±90º (estás mirando totalmente hacia arriba o hacia abajo) el coseno será 0 y no hay componente X.

```
frontVec.z = static_cast<float>(
    sin(glm::radians(yaw)) *
    cos(glm::radians(pitch))
);
```

Para la proyección del eje Z, oscilamos por el Yaw y su seno. Es una relación mucho más directa ya en verdad te da un poco igual el Pitch. Si el Yaw es 0, el seno también lo es y por lo tanto el eje Z es 0 (miras hacia el X positivo). Por otro lado, cuando el Yaw es ±90º, su seno y su eje Z será también ±1 (miras en el eje Z negativo o positivo). Esta es la razón por la que vamos a inicializar el Yaw a -90, en OpenGL el estándar es mirar siempre al eje Z negativo.

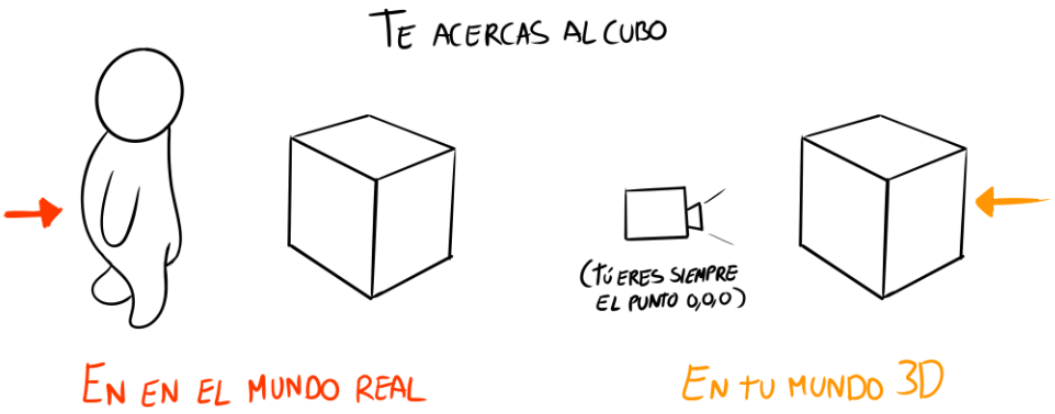
Si controlas de mates, no habrá sido ningún problema esta parte. Si te has mareado un poco, simplemente confía en que los numeritos mágicos están haciendo su trabajo. Ya tenemos los ejes de nuestro vector front, con esto sabremos siempre en qué dirección está mirando nuestra cámara.

4.2.4.3. Mirando por la cámara

Vale, vale, suficientes ángulos. ¿Cómo saco yo la matriz vista de todo esto? Sencillo, las matrices tienen una cosa muy graciosa con la que si tienes tres ejes perpendiculares (como en nuestro caso, el vector front, el right y el up) puedes hacer una matriz con estos y transformar cualquier vector a este espacio de coordenadas multiplicándolo por ella. A esta matriz se le conoce como LookAt.

$$Matriz\ LookAt = \begin{bmatrix} Rx & Ry & Rz & 0 \\ Ux & Uy & Uz & 0 \\ Fx & Fy & Fz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -Px \\ 0 & 1 & 0 & -Py \\ 0 & 0 & 1 & -Pz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Donde R es el vector right, U el up, F el fornt y P es la posición de la cámara. La posición la ponemos en negativo porque, en verdad, la cámara no suele moverse, sino que se mueve el mundo respecto a la cámara. Es decir, si das un paso adelante hacia el cubo en tu mundo 3D, como tu centro del mundo desde tu perspectiva10 es la cámara, el resultado es que en verdad el cubo (y el mundo en general) da un paso en la dirección opuesta a tu movimiento.



Esta matriz es capaz de transformar cualquier vector que multipliques por ella a su espacio de coordenadas y decirte dónde está desde el punto de vista de la cámara. Un momento, ¿transformar cualquier vector a nuestro espacio de

coordenadas? Si revisas el apartado 3.3.2 recordarás que eso es justo lo que necesitamos en la matriz vista. Por suerte, GLM tiene una función para ella.

```
// Método que calcula la matriz vista usando la matriz LookAt
glm::mat4 getViewMatrix() const{
    // Le pasamos la posición, punto al que mira la cámara
    // y el vector up. Con esto GLM construye la matriz.
    return glm::lookAt(position, position + front, up);
};
```

4.2.4.4. Moviendo la cámara

Ya podemos sacar la matriz vista de nuestra cámara y, por lo tanto, mirar por ella. Sin embargo, hasta ahora es una cámara igual de aburrida que la del capítulo anterior, vamos a hacer que se mueva un poco que para algo la hemos hecho entidad.

Lo primero es tener en cuenta que, cuando la movamos y rotemos, la cámara necesitará mantener actualizados sus vectores. Ya sabemos cómo sacar el vector front, con él y el vector worldUp podemos sacar el vector right (será perpendicular a ambos). Una vez tengamos el right y el front, el up como también ha de ser perpendicular a ambos lo podemos sacar igual de fácil.

```
// Actualiza los valores de up y right segun los valores de Euler
void updateCameraVectors(){
    // Variable auxiliar para calcular el vector front
    // antes de normalizarlo
    glm::vec3 frontVec{};

    // Calculamos la componente X, Y y Z del vector front
    // usando los ángulos yaw y pitch
    frontVec.x = static_cast<float>(
      cos(glm::radians(yaw)) * cos(glm::radians(pitch)));
    frontVec.y = static_cast<float>(sin(glm::radians(pitch)));
    frontVec.z = static_cast<float>(
      sin(glm::radians(yaw)) * cos(glm::radians(pitch)));

    // Normalizamos el vector front para que tenga longitud 1,
    // esto es necesario porque los vectores deben ser vectores
    // unitarios de dirección para usarse con LookAt
    front = glm::normalize(frontVec);

    // Calculamos el vector right haciendo producto cruz entre front
    // y worldUp, esto nos da un vector perpendicular a ambos que
    // apunta a la derecha de la cámara.
    right = glm::normalize(glm::cross(front, worldUp));

    // Calculamos el vector up haciendo producto cruz entre right y
    // front, esto nos da un vector perpendicular a ambos que apunta
    // hacia arriba desde la perspectiva de la cámara
    up = glm::normalize(glm::cross(right, front));
};
```

Con los vectores actualizados, nuestra matriz LookAt sabrá seguirle el ritmo a nuestro movimiento. Sin embargo, nosotros no haremos los movimientos, sino nuestro usuario, y lo hará a través de inputs. Vamos a desarrollar cuatro tipos de movimientos comunes: Movimiento directo, movimiento dirigido, rotación con el ratón y zoom.

El movimiento directo es el más sencillo de entender, recibimos una dirección y nos movemos hacia ella. Esto, generalmente, lo habrás hecho en muchos videojuegos pulsando el clásico WASD, aun así, no podemos anclarnos a esas teclas, tenemos que ser más genéricos. Con esa idea prepararemos un enumerado de las direcciones que nos permitirá abstraer un poco.

```
// Para mayor usabilidad y permitir que el movimiento sea
// igual en todos los dispositivos
enum CameraMovement{
    FORWARD, BACKWARD, LEFT, RIGHT
};

// Velocidad de movimiento base de la cámara
float movementSpeed{};

// Aqui el usuario podra mover la camara segun un delta time,
// asegurando un movimiento estable respecto al framerate
// (Luego profundizo un poco más al respecto)
void processKeyboard(CameraMovement direction, float deltaTime){
    float velocity{movementSpeed * deltaTime};
    if (direction == FORWARD)
        position += front * velocity;
    if (direction == BACKWARD)
        position -= front * velocity;
    if (direction == LEFT)
        position -= right * velocity;
    if (direction == RIGHT)
        position += right * velocity;
};
```

El movimiento dirigido también es sencillo, nos pasan un punto en el mundo y hacemos que nuestra cámara apunte hacia él modificando el vector front. Si quisieses hacer que la cámara mirase a tu personaje sería tan fácil como pasarle la posición del mismo.

```
// Posiciona la cámara mirando a una posición en especifico
// Nos aseguramos de normalizar después el vector
void toTarget(glm::vec3 target){
    front = glm::normalize(glm::vec3(
        target.x - position.x,
        target.y - position.y,
        target.z - position.z));
}
```

El movimiento rotatorio usando el ratón tiene un poco más de truco. El usuario nos pasará la diferencia de posición del ratón entre el frame anterior y el nuevo, es decir, cómo se ha movido. Con esto, modificamos los ángulos de Euler.

```
// Guardamos la sensibilidad del ratón
float mouseSensitivity{};

// Si el usuario quiere que la cámara se mueva con el
// ratón, este método se lo permitirá
void processMouseMovement(float xoffset, float yoffset){
    // Multiplicamos el movimiento puro del ratón por la
    // sensibilidad para que sea más lento o más rápido
    xoffset *= mouseSensitivity;
    yoffset *= mouseSensitivity;
    // Añadimos los offsets a los ángulos de Euler
    yaw   += xoffset;
    pitch += yoffset;

    // Limitamos el pitch para evitar el gimbal lock
    if (pitch > 89.0f)
        pitch = 89.0f;
    if (pitch < -89.0f)
        pitch = -89.0f;

    // Actualizamos los vectores con los nuevos ángulos
    updateCameraVectors();
};
```

Por último, el zoom, que también se puede ver como que estás cambiando el FOV (campo de visión). Si recuerdas en el apartado 4.2.3, concretamente en la parte donde dibujamos y creamos la matriz de proyección, usábamos el atributo zoom de la cámara. Si modificamos entonces este atributo, modificaremos el FOV de la cámara, dando la ilusión de que añadimos o quitamos zoom.

```
// Zoom de la camara, por defecto me gusta ponerlo a 45°
float zoom{};

// Este método permitirá al usuario controlar el zoom
// modificando el FOV de la cámara
void processMouseScroll(float yoffset){
    // Restamos el offset al zoom (si el offset es positivo,
    // restamos y si es negativo, sumamos) porque es lo que
    // espera el usuario (abajo = alejar, arriba = acercar)
    zoom -= static_cast<float>(yoffset);
    // Limitamos el zoom minimo a 1 grado para que no se
    // dé la vuelta la matriz de proyección con un zoom negativo
    if (zoom < 1.0f)
        zoom = 1.0f;
    // Limitamos el zoom máximo a 100 grados
    if (zoom > 100.0f)
        zoom = 100.0f;
};
```

Hablo de zoom y no directamente de FOV porque siento que es más intuitivo para el usuario llamarlo zoom. Siéntete libre de llamarlo FOV si quieres ser más correcto.

4.2.4.5. Construyendo la cámara

Con esto ya tenemos todo lo necesario para crear el constructor de la cámara con los valores pasados por el usuario y los valores por defecto. También recuerda que como hereda de E_Entity (la entidad genérica) necesita definir un método draw aunque nunca se dibuje.

```
// Constructor de la cámara, inicializa todo lo necesario
E_Camera(glm::vec3 position = glm::vec3(0.0f, 0.0f, 0.0f),
  glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f),
  float yaw = -90.0f, float pitch = 0.0f)
   : front(glm::vec3(0.0f, 0.0f, -1.0f)),
     movementSpeed(2.0f), mouseSensitivity(0.1f), zoom(45.0f){
    this->position = position;
    // El up de verdad lo calculo después, pero le llamo up en
    // el parámetro porque será más intuitivo para el usuario.
    // Nos aseguramos de que esté normalizado.
    this->worldUp = glm::normalize(up);
    this->yaw = yaw;
    this->pitch = pitch;
    updateCameraVectors();
};

// Método Draw obligatorio por ser una entidad,
// pero una cámara no lo necesita
void draw(glm::mat4 matrix = glm::mat4(1.0f),
      E_Camera* principalCamera = nullptr) override{};
```

Y ya está, con esto ya tienes la cámara de tu mundo 3D y la puedes mover como quieras. Te dije que iba a introducir muchos conceptos nuevos, espero que te haya quedado todo claro. Si necesitas revisar cualquier cosa, te dejo el enlace a la clase[21].

4.3. ADAPTANDO NUESTROS RECURSOS

Bueno, antes de avanzar, ahora que ya tenemos las entidades quizá deberíamos actualizar un poco nuestros recursos para que nuestro gestor de recursos esté al día. Este será un punto rápido.

Como mencionamos en el apartado 4.2.1, usaremos la raíz de nuestro árbol para albergar la cámara principal. Es decir, la que estás usando para ver en la ventana. Esto permite que, como nuestros recursos a veces necesitan información de esta cámara, con guardar el nodo raíz es suficiente, ya que este tendrá la cámara principal. Con esto en mente, necesitaremos modificar la cabecera de nuestro ResourceManager y el recurso genérico R_Resource.

[21] Archivo E_Camera.hpp

```
// Forward declaration
struct Node;

// Gestor de recursos
struct ResourceManager{
    ...
    private:
        ...
        // Puntero a la raiz
        Node* scene{};
};
```

```
struct Node;

// Metodo general de recursos,
que lo heredaran todos los
recursos
struct R_Resource{
    ...
    Node* scene;
};
```

Como puedes ver, el cambio es mínimo, más que nada porque aún no le damos importancia a que el recurso conozca información en la escena, como por ejemplo su posición. Sin embargo, como motor, está bien que exista la posibilidad por si mañana te apetece expandirlo.

4.4. UNA ESCENA CUBISTA Y ORDENADA

¿Qué, cómo te sientes? Has pasado de tener un montón de cubos sueltos a tener recursos y entidades, automatizando todo el proceso de renderizado. ¡Vaya cambio! Ha sido mucha información de golpe, ¿por qué no vamos a lo simple y sólo recreamos la escena del capítulo anterior, pero con nuestra nueva arquitectura?

Vamos allá, el primer paso es parametrizar un poco el código, que no me gusta verlo metido todo en un main.

```
// El color del fondo, para evitar recalcularlo si es el mismo
Color colorFondo;

// Método donde encapsulamos la responsabilidad de limpiar el fondo
void cleanBackground(Color c) {
    // Como las divisiones son costosas, guardamos una copia y
    // solo dividimos si el color es diferente al ultimo
    if (c != colorFondo) {
        // Reasignamos los colores al colorFondo global,
        // en RGBA para facilitarselo al usuario
        colorFondo =
            Color(c.r/255.0f, c.g/255.0f, c.b/255.0f, c.a/255.0f);
    }
    ...
```

```
    ...

    // Limpiamos la pantalla con el color de limpieza
    glClearColor(
      colorFondo.r, colorFondo.g, colorFondo.b, colorFondo.a);
    // Limpiamos el buffer de color y el buffer de profundidad
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}

// Método donde encapsulas la responsabilidad del setup de OpenGL
bool configurarOpenGL() {
    glfwInit();

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    window = glfwCreateWindow(ANCHO, ALTO, NOMBRE, NULL, NULL);
    if (window == NULL) {
        std::cout <<
            "Ha fallado la creacion de la ventana" << std::endl;
        glfwTerminate();
        return false;
    }

    glfwMakeContextCurrent(window);

    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {
        std::cout <<
            "Ha fallado la inicializacion de GLAD" << std::endl;
        return false;
    }

    glViewport(0, 0, ANCHO, ALTO);
    glEnable(GL_DEPTH_TEST);
}
```

Lo único a comentar aquí es que, para hacérselo más intuitivo al usuario, vamos a esperar que nos traiga los colores en RGB y no en escala de OpenGL. También, para optimizarlo, guardamos una copia del color y solo recalculamos en el caso de que el color que nos pasa sea diferente al anterior. Créeme, teniendo en cuenta que estas divisiones van en el bucle de renderizado, lo agradecerás.

Ahora vamos a inicializar nuestro árbol de la escena e instanciar nuestro gestor de recursos, y vamos a empezar a cargar los shaders con el método que implementamos.

```
// La raíz del árbol
std::unique_ptr<Node> sceneRoot{};
// Instancia del gestor de recursos
ResourceManager RM{};
// Vectores para los punteros a los nodos, camaras y modelos
std::vector<std::unique_ptr<Node>> nodes{};
std::vector<std::unique_ptr<E_Camera>> cameras{};
std::vector<std::unique_ptr<E_Model>> models{};
...
```

```
...
int main() {
    // Llamamos a nuestra función que configura OpenGL
    if (!configurarOpenGL())
        return -1;

    // Iniciamos la raíz del árbol de la escena
    sceneRoot = std::make_unique<Node>();
    // Le decimos al Resource Manager que esta es nuestra raíz
    RM.setScene(sceneRoot.get());
    // Cargamos nuestros shaders basicos
    Shader* basicShader = &RM.getShader("base",
      "./Shaders/vertex_shader_basico.vs",
      "./Shaders/fragment_shader_basico.fs")->shader;
```

Sin embargo, mucho me temo que crear nuestra cámara y nuestros cubos va a ser un pelín más peliagudo, así que vamos a encapsular la creación en métodos. No te asustes por el código y el uso de punteros inteligentes, si vas línea a línea verás que no es tan complicado. Comencemos con la cámara.

```
Node* createCamera(
  glm::vec3 position, glm::vec3 up, float yaw, float pitch) {
    // Creamos un nodo como un puntero único
    auto nodeCam = std::make_unique<Node>();
    // Lo pusheamos al vector de nodos y le pasamos la
    // responsabilidad de memoria
    nodes.push_back(std::move(nodeCam));
    // Lo recuperamos como una referencia para trabajar con él
    auto& nCam = nodes[nodes.size() - 1];
    // Creamos la entidad con los parámetros
    auto entityCam = std::make_unique<E_Camera>(
      position, up, yaw, pitch);
    // Lo pusheamos y le pasamos la responsabilidad de memoria
    cameras.push_back(std::move(entityCam));
    // La recuperamos como referencia
    auto& eCam = cameras[cameras.size() - 1];
    // Si no existe ya una, la asigna como cámara principal
    if(sceneRoot->getPrincipalCamera() == NULL){
        sceneRoot->setPrincipalCamera(eCam.get());
    }
    // Le asigna al nodo la entidad
    nCam->setEntity(eCam.get());
    // Lo añadimos a la raíz como hijo, para que esté en la escena
    sceneRoot->addChild(nCam.get());
    // Devuelve un puntero al nodo, para el usuario
    return sceneRoot->getChildren()[
      sceneRoot->getChildren().size() - 1];
}
```

Lo único que igual no te queda claro es eso de “le pasamos la responsabilidad de memoria”. Si recuerdas, al usar un Unique_Ptr si este se eliminaba, la entidad se eliminaba también. Al meter estos punteros en un vector, lo que estamos haciendo es que cuando este vector se destruya, todos los punteros (y por lo tanto sus entidades) lo harán también. De ahí que “le pasemos la responsabilidad”. Para los cubos usaremos una filosofía parecida.

```
Node* createCube(glm::vec3 pos, glm::vec3 size, Color color,
  float rotAngle, glm::vec3 rotAxis) {
    // Creamos el nodo
    auto nodeCube = std::make_unique<Node>();
    // Lo metemos en el vector de nodos
    nodes.push_back(std::move(nodeCube));
    // Lo recuperamos como referencia
    auto& nCube = nodes[nodes.size() - 1];
    // Creamos la entidad, esta vez usando nuestro
    // resource manager como implementamos en su momento
    auto entityModel = std::make_unique<E_Model>(
      &RM, "Cubo", RType::RCube);
    // Lo metemos en el vector de modelos
    models.push_back(std::move(entityModel));
    // Lo recuperamos como referencia
    auto& eCube = models[models.size() - 1];
    // Le metemos nuestro shader, con el nombre que le dimos antes
    eCube->setShader("base");
    // Aplicamos transformaciones para ponerlo donde nos ha dicho
    // el usuario con sus parámetros que lo quiere
    nCube->translate(glm::vec3(pos.x, pos.y, pos.z));
    nCube->rotate(
      glm::vec4(rotAxis.x, rotAxis.y, rotAxis.z, rotAngle));
    nCube->scale(glm::vec3(size.x, size.y, size.z));
    // Le ponemos el color y el aspect ratio
    eCube->color = color;
    eCube->setAspect(aspect);
    // Guardamos la entidad dentro del nodo
    nCube->setEntity(eCube.get());
    // Lo añadimos a la escena
    sceneRoot->addChild(std::move(nCube.get()));
    // Recuperamos el nodo para el usuario
    return sceneRoot->getChildren()[
      sceneRoot->getChildren().size() - 1];
}
```

Como ves, sigue una estela similar, lo único es que como aquí sí estamos creando un recurso, en el constructor de la entidad sí pasamos nuestro resource manager. Con esto, podemos continuar con nuestro main creando una escena como la anterior.

```
// Creamos la cámara principal
Node* camara = createCamera(glm::vec3(-0.3f, 0.0f, 5.0f),
      glm::vec3(0.0f, 1.0f, 0.0f), -90.0f, 0.0f);

// Creamos los cubos, igual que en la Fase2.cpp
std::vector<Node*> cubos{};
Node* cubo1  = createCube(glm::vec3(0.0f, 0.0f, 0.0f),
      glm::vec3(1.0f, 1.0f, 1.0f), Color(1.0f, 0.0f, 0.0f, 1.0f),
      45.0f, glm::vec3(1.0f, 0.0f, 0.0f));
Node* cubo2  = createCube(glm::vec3(2.0f, 5.0f, -15.0f),
      glm::vec3(1.0f, 1.0f, 1.0f), Color(0.0f, 1.0f, 0.0f, 1.0f),
      45.0f, glm::vec3(1.0f, 0.0f, 0.0f));
Node* cubo3  = createCube(glm::vec3(-1.5f, -2.2f, -2.5f),
      glm::vec3(1.0f, 1.0f, 1.0f), Color(0.0f, 0.0f, 1.0f, 1.0f),
      45.0f, glm::vec3(1.0f, 0.0f, 0.0f));
Node* cubo4  = createCube(glm::vec3(-3.8f, -2.0f, -12.3f),
      glm::vec3(1.0f, 1.0f, 1.0f), Color(1.0f, 1.0f, 0.0f, 1.0f),
      45.0f, glm::vec3(1.0f, 0.0f, 0.0f));
...
```

```
...
Node* cubo4  = createCube(glm::vec3(-3.8f, -2.0f, -12.3f),
      glm::vec3(1.0f, 1.0f, 1.0f), Color(1.0f, 1.0f, 0.0f, 1.0f),
      45.0f, glm::vec3(1.0f, 0.0f, 0.0f));
Node* cubo5  = createCube(glm::vec3(2.4f, -0.4f, -3.5f),
      glm::vec3(1.0f, 1.0f, 1.0f), Color(0.0f, 1.0f, 1.0f, 1.0f),
      45.0f, glm::vec3(1.0f, 0.0f, 0.0f));
Node* cubo6  = createCube(glm::vec3(-1.7f, 3.0f, -7.5f),
      glm::vec3(1.0f, 1.0f, 1.0f), Color(1.0f, 0.0f, 1.0f, 1.0f),
      45.0f, glm::vec3(1.0f, 0.0f, 0.0f));
Node* cubo7  = createCube(glm::vec3(1.3f, -2.0f, -2.5f),
      glm::vec3(1.0f, 1.0f, 1.0f), Color(1.0f, 1.0f, 1.0f, 1.0f),
      45.0f, glm::vec3(1.0f, 0.0f, 0.0f));
Node* cubo8  = createCube(glm::vec3(1.5f, 2.0f, -2.5f),
      glm::vec3(1.0f, 1.0f, 1.0f), Color(0.0f, 0.0f, 0.0f, 1.0f),
      45.0f, glm::vec3(1.0f, 0.0f, 0.0f));
Node* cubo9  = createCube(glm::vec3(1.5f, 0.2f, -1.5f),
      glm::vec3(1.0f, 1.0f, 1.0f), Color(0.5f, 0.5f, 0.5f, 1.0f),
      45.0f, glm::vec3(1.0f, 0.0f, 0.0f));
Node* cubo10 = createCube(glm::vec3(-1.3f, 1.0f, -1.5f),
      glm::vec3(1.0f, 1.0f, 1.0f), Color(1.0f, 0.5f, 0.0f, 1.0f),
      45.0f, glm::vec3(1.0f, 0.0f, 0.0f));

// Nos guardamos los cubos para iterar por ellos luego
cubos.push_back(cubo1);
cubos.push_back(cubo2);
cubos.push_back(cubo3);
cubos.push_back(cubo4);
cubos.push_back(cubo5);
cubos.push_back(cubo6);
cubos.push_back(cubo7);
cubos.push_back(cubo8);
cubos.push_back(cubo9);
cubos.push_back(cubo10);
```

Mucho más intuitivo, ¿verdad? Tu main se ha quedado muchisimo más pequeño y simple, como debería ser. Vamos a ver si lo mismo le pasa a nuestro bucle de renderizado. Pero antes, tengo una deuda contigo, te dije que te profundizaría sobre algo llamado “DeltaTime”. Llegó el momento.

4.4.1. El Delta Time

Pensemos en el siguiente caso, quieres mover la cámara hacia donde está mirando y haces el siguiente código:

```
cameraPosition += cameraFront * 0.1f;
```

Haces unas pruebas en tu PC turbo gamer rgb y va perfecto, movimiento fluido y todo guay. Le pasas tu aplicación a tu colega el que no encuentra trabajo y sobrevive a base de sopa de hielo y te dice que vaya mierda, que la cámara va muy lento. ¿Qué ha podido pasar? Bueno, el bucle de renderizado no espera a nadie, si en tu PC tu programa va a 300 FPS entonces la cámara va a 300x0.1, mientras que a tu colega que le corre a 30 irá a 30x0.1. Ahí está el problema.

¿Cómo lo solucionamos? Pues la cámara de tu colega debería ir ligeramente más rápida que la tuya, para compensar. ¿Y cuánto más? Para saber cuánto es para lo que necesitamos el Delta Time.

El Delta Time se define como el tiempo que has tardado en reproducir el anterior frame. Sólo tenemos que multiplicar toda velocidad o movimiento por este Delta Time. Piénsalo así, si tu PC gamer tarda 0.016 segundos en calcular un frame (su Delta Time) entonces el cálculo la velocidad sería el siguiente:

```
cameraPosition += cameraFront * 0.1f * 0.016;
```

Por otro lado, el de tu colega tiene un Delta Time de 0.033 (ha tardado más), entonces calcularíamos la velocidad así:

```
cameraPosition += cameraFront * 0.1f * 0.033;
```

El resultado será mayor en el PC de tu colega que en el tuyo, sin embargo, la diferencia será sólo la necesaria para que ambos ordenadores estén corriendo a la misma velocidad. Ahora que hemos entendido esto, vamos a ver cómo lo puede calcular nuestra aplicación.

```
// Variables para deltaTime
float lastFrame{};
float deltaTime{};

int main() {
    ...

    // Inicializamos el bucle de renderizado
    while (!glfwWindowShouldClose(window)) {
        ...
        // Calculamos el deltaTime
        float currentFrame = static_cast<float>(glfwGetTime());
        deltaTime = currentFrame - lastFrame;
        lastFrame = currentFrame;
        ...
    }
}
```

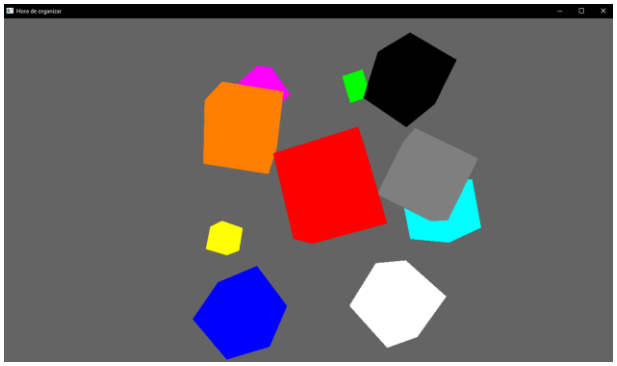
Como ves, sencillito y para toda la familia. Era más complicado entender qué hace y por qué es importante tenerlo en cuenta que calcularlo. Lo único que tendremos que tener cuidado porque al multiplicar por décimas de segundo, todas nuestras velocidades van a tener que ser un par de magnitudes más altas para compensar esa multiplicación.

4.4.2. El bucle de renderizado ordenado

Vale, ya tenemos configurado OpenGL, el gestor de recursos instanciado, el árbol creado y nuestros diez cubos como entidades dentro del mismo. Ahora vamos a ver cómo hacemos nuestro bucle de renderizado. Vamos a tener en cuneta el siguiente flujo: Borramos el fondo>Calculamos Delta Time>Rotamos los cubos>Recorremos el árbol para dibujarlos>Eventos/Buffers de OpenGL

```
// Inicializamos el bucle de renderizado
while (!glfwWindowShouldClose(window)) {
    // Limpiamos el fondo
    cleanBackground(Color(100.0f, 100.0f, 100.0f, 255.0f));

    // Calculamos el deltaTime
    float currentFrame = static_cast<float>(glfwGetTime());
    deltaTime = currentFrame - lastFrame;
    lastFrame = currentFrame;

    // Iteramos por cada cubo
    for (unsigned int i = 0; i < 10; i++) {
        // Lo mismo que la fase2,
        // pero usando los nodos de nuestro árbol
        Node* cubo = cubos.at(i);
        float frameAngle{};
        float rotationSpeed{40.0f};
        if (i > 0) {
            // Tenemos en cuenta el Delta Time
            frameAngle = deltaTime *
                  rotationSpeed * static_cast<float>(i);
        } else {
            frameAngle = deltaTime * rotationSpeed;
        }
        cubo->rotate(glm::vec4(1.0f, 0.3f, 0.5f, frameAngle));
    }

    // Recorremos el árbol
    sceneRoot->traversal(glm::mat4{ 1.0f },
      sceneRoot->getPrincipalCamera());

    glfwPollEvents();
    glfwSwapBuffers(window);
}
```

Si lo ejecutas, verás una escena nostálgicamente similar.



4.4.3. Moviéndonos por la escena

Ya tenemos nuestra escena igualita que en el capítulo anterior, pero con una arquitectura más sólida. Pero yo te prometí una cámara que se movía y te la voy a dar. Vamos a procesar el input de los usuarios.

Vamos a recibir tres tipos de input concretamente, por teclado y por ratón. Este último lo podremos mover libremente o hacer zoom con la rueda. Empecemos por el primero, es muy fácil leer el input del teclado porque si recuerdas bien queríamos justo GLFW para ese trabajo, así que él nos echa una mano.

```
// Procesamos los inputs del teclado de los usuarios con GLFW
void processInput() {
    // El movimiento de la cámara con WASD,
    // usando los enumerados de GLFW y glfwGetKey
    if(glfwGetKey(window, GLFW_KEY_W)) {
        // Recuperamos la cámara de nuestro vector y
        //usamos nuestro método de movimiento
        cameras.at(0)->processKeyboard(FORWARD, deltaTime);
    }
    if(glfwGetKey(window, GLFW_KEY_S)) {
        cameras.at(0)->processKeyboard(BACKWARD, deltaTime);
    }
    if(glfwGetKey(window, GLFW_KEY_A)) {
        cameras.at(0)->processKeyboard(LEFT, deltaTime);
    }
    if(glfwGetKey(window, GLFW_KEY_D)) {
        cameras.at(0)->processKeyboard(RIGHT, deltaTime);
    }
    // Escape para cerrar la ventana.
    if(glfwGetKey(window, GLFW_KEY_ESCAPE)) {
        // Con esto, saldrá del bucle de renderizado
        glfwSetWindowShouldClose(window, true);
    }
}
```

Bien, ahora a por el movimiento del ratón. Es un poco más complejo, necesitamos, si lo recuerdas del apartado 4.2.4.4, la diferencia de posición en el eje X e Y del ratón. Nos apoyamos también en GLFW para recibir la posición actual del ratón, pero no sólo la diferencia la calculamos nosotros, sino que tenemos que avisarle a GLFW para que nos devuelva la posición del ratón cada vez que se mueva. Para eso tenemos que añadir en setupOpenGL lo siguiente:

```
glfwSetCursorPosCallback(window, movimientoRaton);
```

Con esto, GLFW llamará a nuestra función movimientoRaton cada vez que el ratón sea movido por el usuario, pasándole las posiciones actuales en ambos ejes como parámetro. Con esto, tenemos todo lo necesario.

```
// Guardamos las posiciones anteriores del ratón
float lastX{}, lastY{};
// Por si es la primera lectura del ratón
bool firstMouse{ true };

// Procesamos los inputs del movimiento del ratón
void movimientoRaton(GLFWwindow* window,
  double xPosIn, double yPosIn) {
    // Guardamos la posición que nos pasa GLFW
    float xPos{ static_cast<float>(xPosIn) };
    float yPos{ static_cast<float>(yPosIn) };

    // Si es la primera vez que leemos el ratón, la
    // guardamos antes de actualizarlas (evitamos nullpointer)
    if (firstMouse) {
        lastX = xPos;
        lastY = yPos;
        firstMouse = false;
    }

    // Calculamos la diferencia entre la posición actual
    // y la anterior en ambos ejes
    float xOffset{ xPos - lastX };
    float yOffset{ lastY - yPos };
    // Actualizamos las posiciones
    lastX = xPos;
    lastY = yPos;

    // Recuperamos la cámara principal, en el caso de que haya
    if (cameras.at(0) != NULL)
        // Con la diferencia entre posiciones,
        //llamamos a nuestro método de movimiento del ratón
        cameras.at(0)->processMouseMovement(xOffset, yOffset);
}
```

Aunque técnicamente tu cámara ya se podría mover el ratón, si lo probases te darías cuenta de que tu ratón al moverlo se sale de la pantalla. Eso es porque no está fijo en la ventana, como suelen hacer los videojuegos por ejemplo. Para ello, debajo de la línea que metimos en setupOpenGL ponemos esta otra:

```
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```

Con esta línea, desactivamos el cursor en la pantalla, lo que permitirá que no se salga de nuestra ventana.

Ahora nos queda el último input, el zoom. Es sencillo también, no tiene mucho misterio. Aunque al igual que con el movimiento del ratón vamos a tener que añadir otra línea a setupOpenGL para que cuando el usuario gire la rueda del ratón, GLFW nos informe del nuevo valor.

```
glfwSetScrollCallback(window, zoomRaton);
```

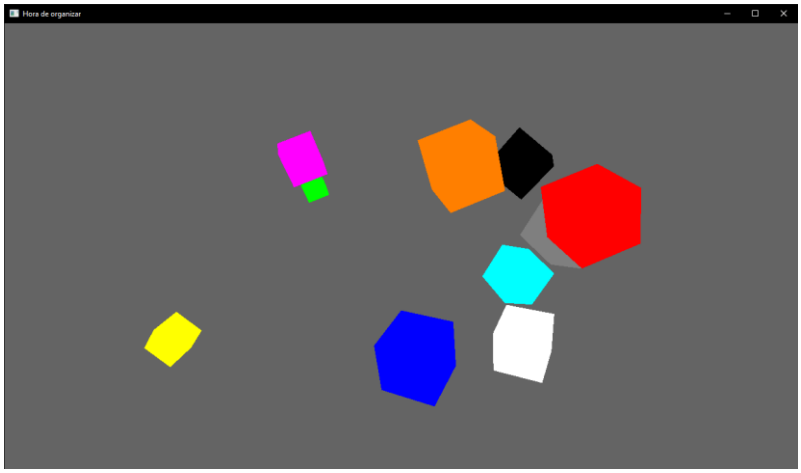
Luego es tan sencillo como coger ese valor y pasárselo a nuestro método.

```
// Procesamos el input de la rueda del ratón, necesita
// X e Y por compatibilidad con dispositivos raros
void zoomRaton(GLFWwindow* window, double xOffset,
  double yOffset) {
    // Si tiene cámara principal, la usamos
    if (cameras.at(0) != NULL)
        // A nosotros obviamente sólo nos interesa
        // el movimiento vertical
        cameras.at(0)->processMouseScroll(
            static_cast<float>(yOffset));
}
```

Perfecto, sólo nos queda un último detalle. Los métodos del ratón, como los hemos configurado con nuestro amigo GLFW, se harán automáticamente cuando el usuario use el ratón. Sin embargo, processInput, el del teclado, no lo hará. Necesitamos llamar al método en el bucle de renderizado, yo te recomiendo arriba del todo.

```
// Inicializamos el bucle de renderizado
while (!glfwWindowShouldClose(window)) {
    // Controlamos el input del usuario
    processInput();
```

Ahora, si lo ejecutas, podrás moverte libremente por la escena. ¡Siente la libertad, te la has ganado!



Prueba a experimentar con los valores por defecto de la cámara, los cubos creados, los movimientos de la cámara como el toTarget. Has creado tu propio escenario y eres libre de jugar con él a tu gusto, y de paso familiarizarte con todo lo que ofrece de momento tu motor. Como siempre te dejo todo el código fuente de este capítulo[22] para que lo revises si algo no te va como debería. Esta vez te lo recomiendo encarecidamente porque para no hacer el capítulo más pesado e ignorado algunos cambios con los imports, dependencias, etc.

[22] Código fuente Fase 3

5. ILUMINANDO LA ESCENA

Ya tienes las bases de un buen motor gráfico, excelente. Sin embargo, tu motor es un poco soso, los cubos son planos y sin gracia. Vamos a darle algo de luz y color a tu escena cubista. Vamos a tratar los tres tipos de luces, las texturas y cómo afecta todo esto a nuestro motor y, sobre todo, a nuestros shaders. De momento no nos vamos a mover de la escena cubista, sólo la vamos a hacer más bonita.

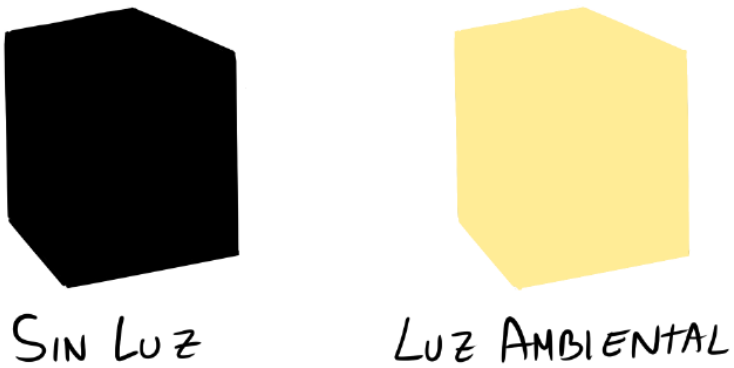
5.1. LA LUZ DE PHONG

Gracias a tus esfuerzos en el capítulo anterior, crear luces será “tan fácil” como crear una nueva entidad. Sin embargo, para dimensionar en verdad todos los cambios que tendremos que hacer en nuestro programa y en los shaders, lo primero será entender cómo funciona la luz en nuestro mundo de OpenGL.

La luz en el mundo real rebota, se absorbe, se dispersa... sería complicado si intentásemos replicar directamente modelando fotón a fotón. Por eso hay muchos modelos que se intentan sólo aproximar el funcionamiento de la luz. Para tu motor vamos a usar el modelo de Phong, que es el más clásico de todos, si quieres sacarle partido a tu RTX5090 con raytracing te animo a hacerlo una vez hayas aprendido las bases con Phong y ya construyas a partir de ahí. El modelo de Phong descompone la luz en tres componentes:

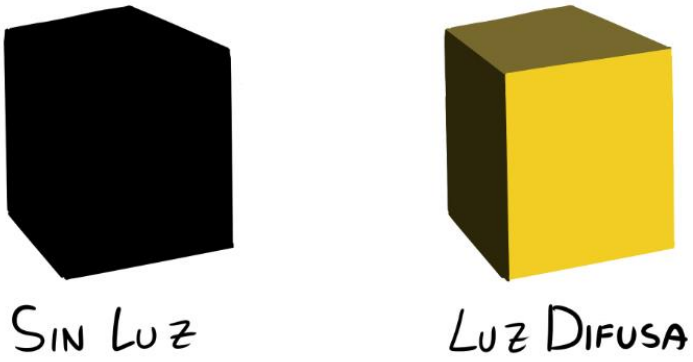
5.1.1. Luz Ambiental

Es aquella que rebota de todas partes, tanto que ya no sabes de dónde viene. Como no tiene una dirección concreta, es como si chocase contra todas las partes del objeto, simulando una luz global.

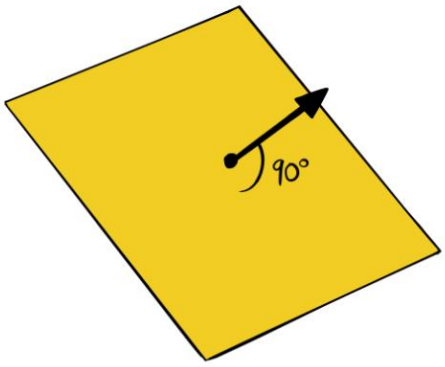


5.1.2. Luz Difusa

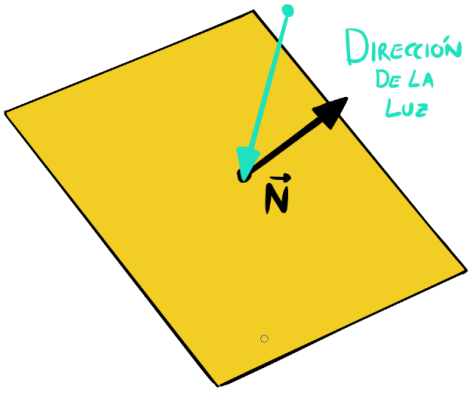
Esta es la que da volumen. Depende del ángulo y la dirección de la luz, es decir, si te da de frente es más brillante que si te da de lado.



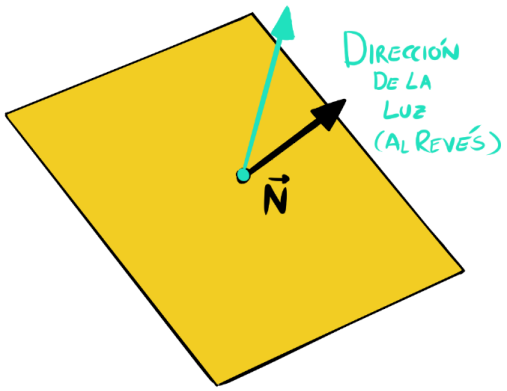
Como ves, necesitamos conocer el ángulo con el que incide la luz. Para poder calcularlo necesitaremos saber primero el vector unitario perpendicular a nuestros vértices o superficies, lo que se conoce como la normal.



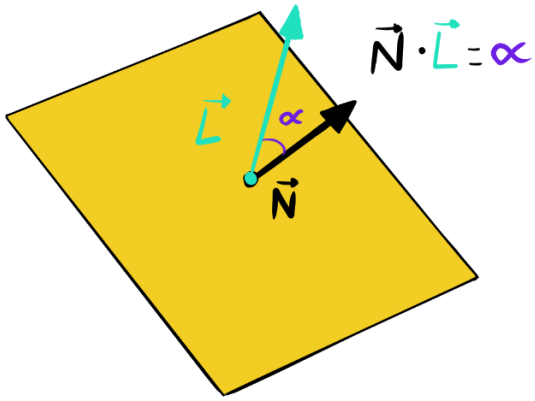
Para la luz difusa, simplemente necesitamos el ángulo que hacen la normal del punto con la dirección de la luz. Imagina que la normal del plano de arriba recibe la siguiente luz, en vector unitario también:



Si le damos la vuelta al vector, tenemos el vector unitario desde la superficie a la fuente de luz:



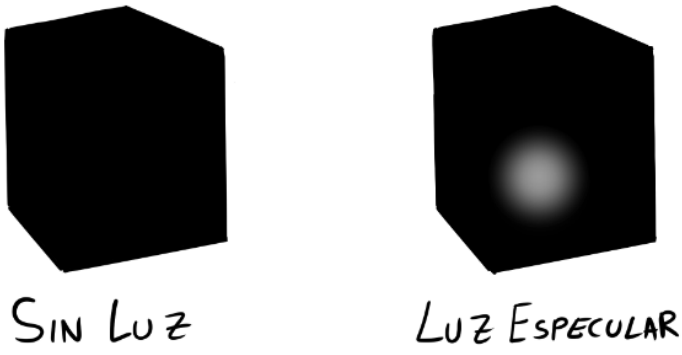
Ya casi lo tenemos, ahora sacando el producto escalar entre estos dos vectores tendremos el ángulo con el que está incidiendo esa luz en nuestro punto:



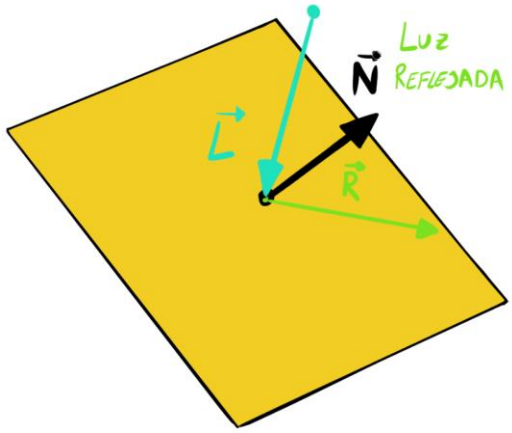
¡Esto era justo lo que necesitamos! Con esto podremos saber el ángulo con el que impacta la luz sobre nuestro punto, pudiendo decidir la intensidad con la que lo vamos a iluminar. Y en el caso de ser un ángulo negativo, sabremos que la luz le viene por detrás y por lo tanto no se va a iluminar.

5.1.3. Luz Especular

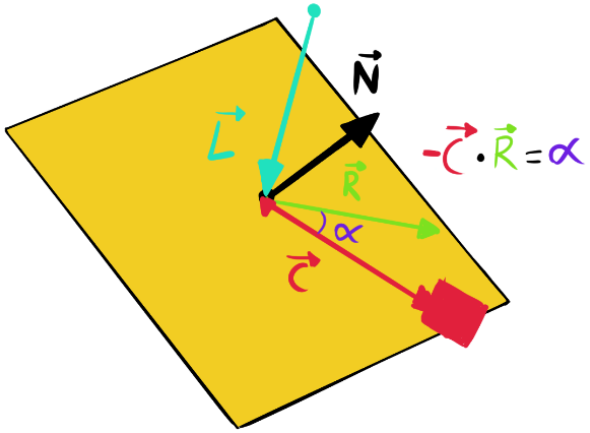
Aquí marcamos el brillo de los objetos. Depende tanto de la dirección de la cámara como del material del objeto. Claro, imagina un objeto metálico, este brillará mucho más que cualquier otro objeto opaco.



Para este caso necesitamos calcular el ángulo que hace nuestra perspectiva desde la cámara con, ojo, el reflejo que hace la luz sobre la superficie.

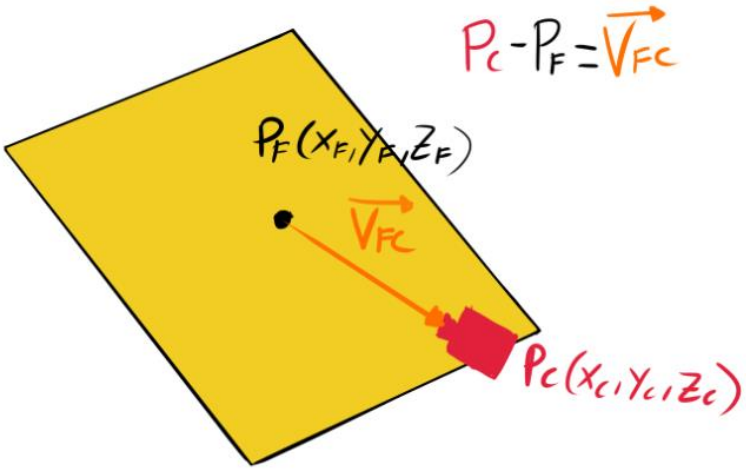


No te preocupes, OpenGL se encargará de calcular ese vector de reflejo, pero es importante que te quedes con que el ángulo que nos importa para calcular la intensidad de la luz especular es el ángulo que hace este con la cámara. Al fin y al cabo, cuánto y cómo brilla un objeto depende de desde dónde lo miras.



Le damos la vuelta también al vector de la cámara para sacar el ángulo como hicimos antes y ya lo tenemos. Sin embargo, aunque matemáticamente esto es así, al contrario que con la luz no tenemos un vector unitario que nos diga desde dónde está mirando la cámara a la superficie (no pienses en el front, eso sólo funcionaría para los vértices que están estrictamente en frente, no el resto).

En verdad es sencillo de sacar este vector dirección, para hacerlo tenemos que sacar la distancia entre la posición el punto de nuestra superficie y la posición de la cámara. ¡De primero de mates!



Siendo Pc el punto de la cámara y Pf el punto de la superficie. Ahora sí, ya tenemos todo lo que necesitamos para sacar el ángulo que forma nuestro reflejo con la cámara y saber la intensidad de nuestro ángulo especular.

5.1.4. Combinado de Phong

El truco no sólo es conocer los tres componentes de la luz, está en combinar sus efectos para obtener una iluminación suave y realista.

Este es el modelo que consigue un resultado más sólido sin irse de madre con la complejidad. Ahora vamos a ver cómo traducimos todo esto de las normales a código de verdad. Y de paso, por qué no, metámonos ya con las texturas.

5.2. CALCULANDO NORMALES Y EXTRAÑAS

Vamos a aprovechar que vamos a tocar nuestros vértices y nuestros cubos y no sólo vamos a sacar las normales, vamos a sacar también nuestras texturas, aunque no te emociones aún no aprenderemos a cargarlas. Te queda un poquito más para eso.

Lo primero que necesitamos es tocar la clase Vertex de R_Mesh. Si recuerdas, te dije que lo ampliaríamos más adelante.

```
struct Vertex{
    glm::vec3 position{};
    glm::vec3 normal{};
    glm::vec2 texCoord{};
};
```

Le añadimos otro vector 3 para las normales, y un vector 2 para las texturas. Recuerda que la normal es un vector unitario de dirección, así que tendrá que tener tantos componentes como ejes tenga nuestro sistema (X, Y y Z en nuestro caso). El vector de texturas, sin embargo, representan un punto en una imagen 2D (por eso le llamamos texCoord) y por lo tanto sólo necesita dos ejes (X, Y).

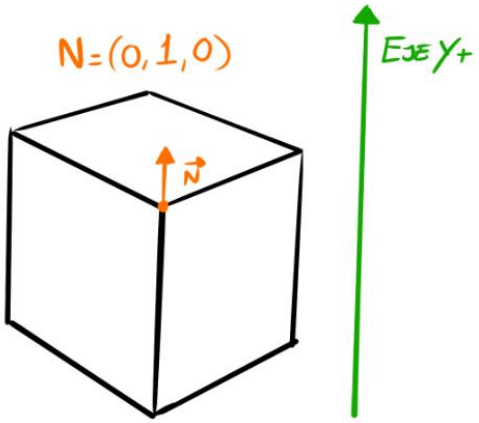
Ahora puff, nos toca volver a calcular los buffers y... un momento, es verdad, si los configuramos usando el tamaño del objeto Vertex en su momento, sólo tenemos que configurar el VAO para indicarle nuestra nueva configuración.

```
// Indicamos el orden en el que va a recibir el VAO la
// información para que luego lo pase por el layout al shader

// Positions
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
      sizeof(Vertex), (void*)0);

// Normales, con un offset ya que no es la primera propiedad de
// la clase Vertex, offsetof te lo calcula automáticamente
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
      sizeof(Vertex), (void*)offsetof(Vertex, normal));

// Textures, con un offset también
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
      sizeof(Vertex), (void*)offsetof(Vertex, texCoord));
```

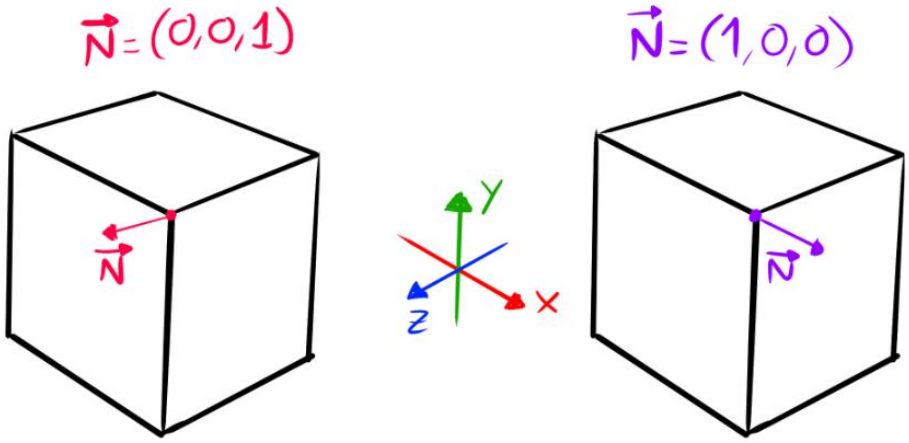
Con esto, nuestro Vertex Shader recibirá tres variables en su layout, así que las vamos a ir definiendo ya.

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;
```

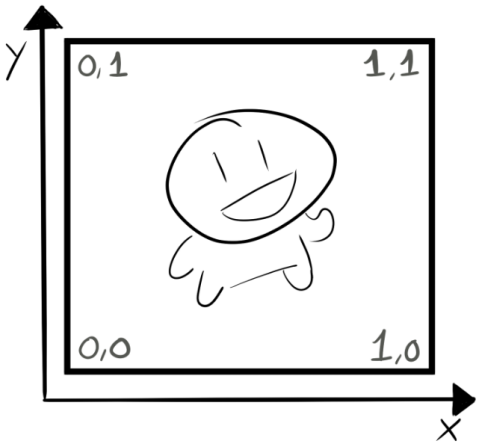
Luego volvemos a los shaders, de momento centrémonos en informar estas normales para nuestros cubos. Si recuerdas bien, las posiciones de los vértices lo hacíamos en el processCube de R_Model. Ahora estos vértices necesitan que les pasemos las normales. Es fácil, te explico, imagina un vértice superior:



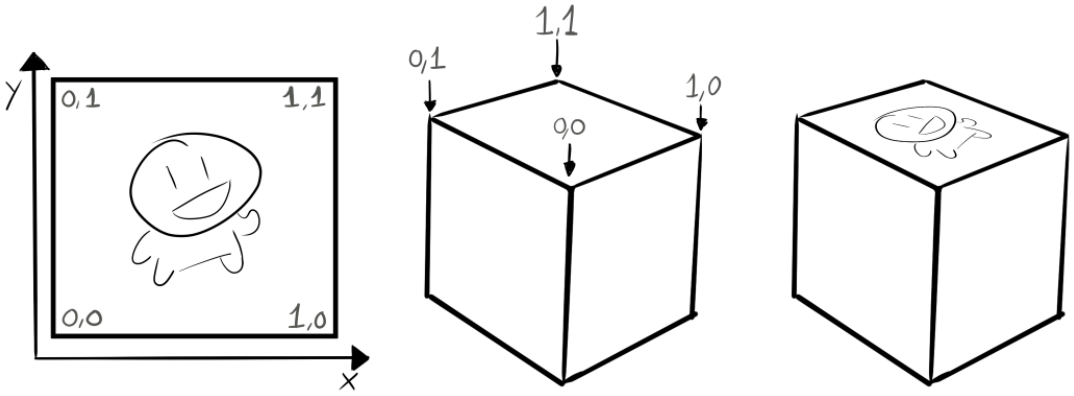
Como la normal es sólo un vector unitario de dirección, sólo necesitas saber el eje en el que está apuntando ese vértice. El problema surgirá cuando dibujemos ese mismo vértice, pero para las otras caras, ya que la normal apuntará en otra dirección (recuerda que la optimización de los índices nos permitía no guardar varias veces el mismo vértice en GPU, pero dibujarlos, como tal, sí que nos tocaba hacerlo varias veces por vértice).



Oh, oh. ¿Esto significa que a tomar viento nuestra optimización de los índices? Me temo que, de momento, para nuestro cubo, sí. Nos toca guardar cada vértice varias veces para poder diferenciar en los shaders sus normales. ¿Pero y las texturas? Como dije antes, sólo son las coordenadas 2D de la imagen.



Si quieres saber en qué punto de la imagen está tu vértice para, por ejemplo, hacer que la cara superior de tu cubo tenga esta textura, simplemente tienes que decirles a tus vértices qué punto de la imagen les corresponde. Luego calcularemos el "Vale, si este punto está aquí y este otro aquí, pintamos esto así", pero de momento con que te quedes con la idea general me sirve.



Muy bien, ya tenemos el espacio en memoria para nuestras normales y nuestras coordenadas de las texturas, ahora nos tocará reformular nuestro processCube para añadirlas. Si entendiste bien cómo funcionaba para la posición, no te será dificil entender el código.

```
// Cara superior (y = 0.5)
vertices.push_back(Vertex{ glm::vec3(-0.5f,  0.5f, -0.5f),
      glm::vec3( 0.0f,  1.0f,  0.0f), glm::vec2(0.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3( 0.5f,  0.5f, -0.5f),
      glm::vec3( 0.0f,  1.0f,  0.0f), glm::vec2(1.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3( 0.5f,  0.5f,  0.5f),
      glm::vec3( 0.0f,  1.0f,  0.0f), glm::vec2(1.0f, 0.0f) });
vertices.push_back(Vertex{ glm::vec3(-0.5f,  0.5f,  0.5f),
      glm::vec3( 0.0f,  1.0f,  0.0f), glm::vec2(0.0f, 0.0f) });

// Cara frontal (z = 0.5)
vertices.push_back(Vertex{ glm::vec3(-0.5f, -0.5f,  0.5f),
      glm::vec3( 0.0f,  0.0f,  1.0f), glm::vec2(0.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3( 0.5f, -0.5f,  0.5f),
      glm::vec3( 0.0f,  0.0f,  1.0f), glm::vec2(1.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3( 0.5f,  0.5f,  0.5f),
      glm::vec3( 0.0f,  0.0f,  1.0f), glm::vec2(1.0f, 0.0f) });
vertices.push_back(Vertex{ glm::vec3(-0.5f,  0.5f,  0.5f),
      glm::vec3( 0.0f,  0.0f,  1.0f), glm::vec2(0.0f, 0.0f) });

// Cara derecha (x = 0.5)
vertices.push_back(Vertex{ glm::vec3( 0.5f, -0.5f,  0.5f),
      glm::vec3( 1.0f,  0.0f,  0.0f), glm::vec2(0.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3( 0.5f, -0.5f, -0.5f),
      glm::vec3( 1.0f,  0.0f,  0.0f), glm::vec2(1.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3( 0.5f,  0.5f, -0.5f),
      glm::vec3( 1.0f,  0.0f,  0.0f), glm::vec2(1.0f, 0.0f) });
vertices.push_back(Vertex{ glm::vec3( 0.5f,  0.5f,  0.5f),
      glm::vec3( 1.0f,  0.0f,  0.0f), glm::vec2(0.0f, 0.0f) });

...
```

```
...
// Cara izquierda (x = -0.5)
vertices.push_back(Vertex{ glm::vec3(-0.5f, -0.5f, -0.5f),
      glm::vec3(-1.0f,  0.0f,  0.0f), glm::vec2(0.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3(-0.5f, -0.5f,  0.5f),
      glm::vec3(-1.0f,  0.0f,  0.0f), glm::vec2(1.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3(-0.5f,  0.5f,  0.5f),
      glm::vec3(-1.0f,  0.0f,  0.0f), glm::vec2(1.0f, 0.0f) });
vertices.push_back(Vertex{ glm::vec3(-0.5f,  0.5f, -0.5f),
      glm::vec3(-1.0f,  0.0f,  0.0f), glm::vec2(0.0f, 0.0f) });

// Cara inferior (y = -0.5)
vertices.push_back(Vertex{ glm::vec3( 0.5f, -0.5f, -0.5f),
      glm::vec3( 0.0f, -1.0f,  0.0f), glm::vec2(0.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3(-0.5f, -0.5f, -0.5f),
      glm::vec3( 0.0f, -1.0f,  0.0f), glm::vec2(1.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3(-0.5f, -0.5f,  0.5f),
      glm::vec3( 0.0f, -1.0f,  0.0f), glm::vec2(1.0f, 0.0f) });
vertices.push_back(Vertex{ glm::vec3( 0.5f, -0.5f,  0.5f),
      glm::vec3( 0.0f, -1.0f,  0.0f), glm::vec2(0.0f, 0.0f) });

// Cara trasera (z = -0.5)
vertices.push_back(Vertex{ glm::vec3( 0.5f, -0.5f, -0.5f),
      glm::vec3( 0.0f,  0.0f, -1.0f), glm::vec2(0.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3(-0.5f, -0.5f, -0.5f),
      glm::vec3( 0.0f,  0.0f, -1.0f), glm::vec2(1.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3(-0.5f,  0.5f, -0.5f),
      glm::vec3( 0.0f,  0.0f, -1.0f), glm::vec2(1.0f, 0.0f) });
vertices.push_back(Vertex{ glm::vec3( 0.5f,  0.5f, -0.5f),
      glm::vec3( 0.0f,  0.0f, -1.0f), glm::vec2(0.0f, 0.0f) });
```

Si te cuesta entender estos números, lo mejor que puedes hacer es con libreta y lápiz dibujar un cubo e intentar representar estas normales y coordenadas visualmente como he hecho yo arriba. Verás como al lograr hacerlo tú a mano, de pronto todo cobra sentido (jeje, en esa idea me inspiré para este libro). Ya que estamos aquí, no me gustan los índices como una hilera de push backs, así que me voy a detener un momento para reducirlo a un bucle sencillo.

```
// Indices para cada cara (2 triángulos por cara, 6 caras en total)
for (int i = 0; i < 6; i++) {
    int base = i * 4;
    indices.push_back(base + 0);
      indices.push_back(base + 1);
      indices.push_back(base + 2);
    indices.push_back(base + 0);
      indices.push_back(base + 2);
      indices.push_back(base + 3);
}
```

Ale, mucho mejor. Ahora volvamos a los shaders, que ahora tienen nueva información con la que trabajar. La luz es compleja, vamos a ir preparando el terreno en el Vertex Shader primero porque los shaders van a escalar en dificultad exponencialmente.

Realmente sí que hay formas de mantener la optimización de los índices como los atlas de texturas o shaders especializados, pero o tienen limitaciones o son realmente complejas y se salen del scope de este proyecto.

5.3. PREPARANDO EL VERTEX SHADER

Vamos a preparar el Vertex Shader primero. Con la información que ahora tiene en sus layouts podemos hacer muchas cosas, pero principalmente será informar al Fragment Shader. Ten en cuenta que la luz y las texturas son cosas del color, estás decidiendo con qué color y con qué intensidad vas a pintar cada fragmento del objeto, lo cual es trabajo de tu Fragment Shader. En concreto, necesitamos pasarle la coordenada de la textura del vértice, la normal y la posición del fragmento a pintar.

```
out vec2 TexCoord;
out vec3 Normal;
out vec3 FragPos;
```

Todos estos valores los calcularemos, como es en el Vertex Shader, una vez por vértice. Quizá te preguntes entonces cómo sabe un fragmento aleatorio dentro de nuestro cubo qué valor debería tener, eso es que has olvidado algo muy importante que explicamos en el 3.5.1, la interpolación. OpenGL calculará todos los puntos intermedios entre los vértices por ti, y eso será con lo que trabajará el Fragment Shader. Dicho esto, vamos ver cómo informamos cada una de esas variables. La primera es la más directa, no necesita ninguna transformación.

```
// Pasamos las coordenadas de texturas
TexCoord = aTexCoord;
```

Con las normales no vamos a tener ese privilegio. Para esto necesitamos la matriz modelo, sólo esa, para conocer el punto del vértice en el mundo. Recuerda que la matriz ModeloVistaProyección nos dice dónde está el punto desde nuestra perspectiva, no la del objeto en sí, de eso se encarga sólo la modelo. Una de dos, o deshacemos la optimización que hicimos en el capítulo anterior y volvemos a pasar las tres matrices por separado, o simplemente pasamos otra vez la matriz modelo como uniform.

```
// Mantenemos el uniform optimizado
uniform mat4 modelViewProjection;
// Pasamos otra vez la modelo pero sola
uniform mat4 model;

void main(){
    // Aquí nos seguimos librando de multiplicar matrices
    gl_Position = modelViewProjection * vec4(aPos, 1.0f);
    // Calculamos las normales por cada vértice
    Normal = mat3(transpose(inverse(model))) * aNormal;
    ...
```

No te asustes, te explico qué ha pasado en ese cálculo de las normales. El punto es que necesitas calcular la dirección de la normal en respecto al mundo, por eso la multiplicamos por la matriz modelo, sin embargo, como tu modelo podría estar escalado de forma no uniforme (imagina que la escalases a Y=0.5) esa multiplicación directa podría fallar. La solución matemática es la siguiente:

$$n' = (M^{-1})^T \cdot n$$

Donde M es la matriz modelo, T la transpuesta y n la normal que le hemos pasado desde processCube. El truco está en que, si no tiene escalado, la inversa de M y su transpuesta se cancelan, dejando a n sin modificar, pero en el caso de que sí que haya un escalado uniforme la inversa de M invierte el efecto de escalado y la transpuesta corrige la dirección, dándonos una n que apunta donde debe apuntar. De nuevo, si da igual las vueltas que le des no entiendes nada, confía en los números mágicos.

Ahora sólo nos queda la posición del fragmento. Esta es más sencilla, sólo necesitamos informarle de la posición del vértice respecto al mundo para que luego OpenGL interpole las posiciones de los fragmentos también respecto al mundo. Por suerte, antes ya nos trajimos la matriz modelo, así que es fácil de calcular.

```
// Le pasamos la posicion del fragment respecto al mundo
FragPos = vec3(model * vec4(aPos, 1.0f));
```

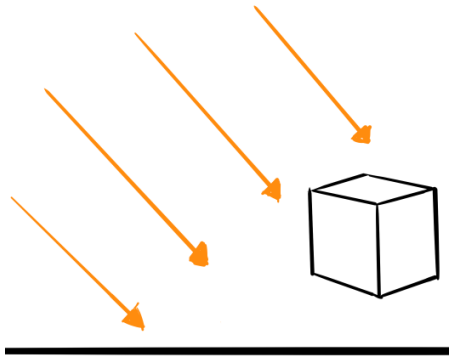
Ahora ya tenemos un Vertex Shader listo para la acción de las luces y las texturas. Vamos a dejar el Fragment Shader para otro momento porque... bueno... ese va a ser un pelín más complicado. De momento, vamos a definir qué tipos de luces tendrá nuestro motor.

5.4. EL SOL, LA BOMBILLA Y LA LINTERNA

Con Phong hemos dicho que la luz se dividía en tres componentes, pero también vamos a tener distintos tipos de fuentes de luz. Eso ya son palabras mayores, estamos hablando de una entidad Luz que se pueda renderizar en tu escena y que, dependiendo de su tipo, tenga diferentes comportamientos en tu shader. Antes de bajar al código, vamos a entender conceptualmente los tres tipos de fuentes de luz con los que vamos a trabajar.

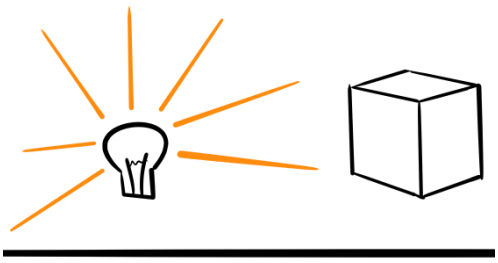
5.4.1. Luz Direccional

Puedes imaginar la luz direccional como la luz del sol, viene de algún punto alejado del espacio y afecta a toda la escena por igual. Lo único que necesitas para esta luz es saber la dirección desde donde viene.



5.4.2. Luz Puntual

Esta la puedes imaginar como una bombilla, nace en algún punto de nuestra escena y emite luz en todas direcciones a partir de ese punto. Pero no sólo eso, al igual que con una bombilla en la vida real, cuanto más cerca estés de ella con más intensidad te dará la luz. Si estás lo suficientemente lejos, de hecho, ni te iluminará. A esto le llamamos atenuación, y se la podremos configurar.



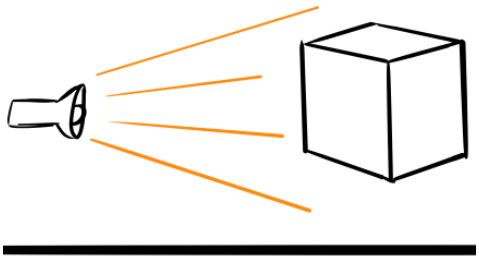
La atenuación se divide en tres coeficientes: La constante, la lineal y la cuadrática. La fórmula para calcularla es la siguiente:

$$\frac{1}{Const + (Lin * D) + (Cuad * D^2)}$$

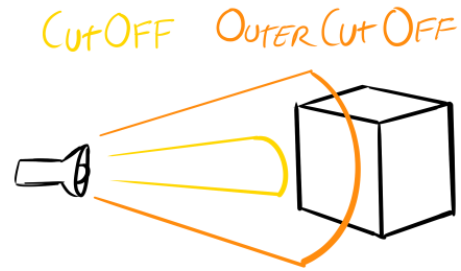
Donde D es la distancia a la que se encuentra la fuente de luz. Estos coeficientes serán modificables por el usuario, permitiendo tener diferentes bombillas con diferentes atributos.

5.4.3. Luz Focal

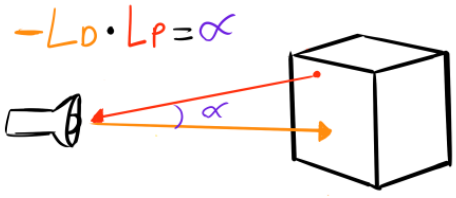
A la luz focal me gusta verla como una linterna. Igual que la puntual, está colocada en alguna parte del mundo, pero no alumbra en todas direcciones, sino que apunta en alguna específica. También se atenúa con la distancia. Necesitaremos conocer además qué tan ancho o estrecho es el foco.



Ya sabemos calcular la atenuación, sin embargo, para saber qué tan ancho es el foco de nuestra linterna es un poco más complejo que aplicar una fórmula. Primero tenemos que entender que un foco se divide en verdad en dos conos:



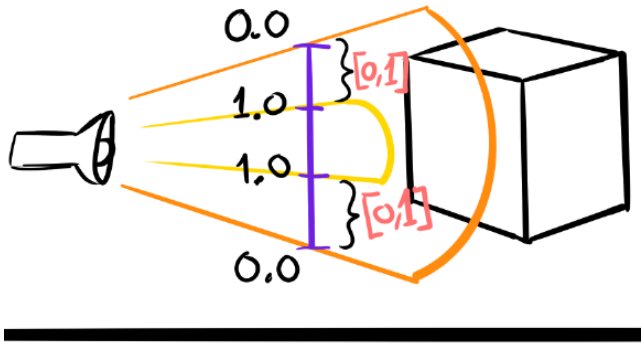
El cono CutOff es el cono central, este siempre dará la luz a su máxima intensidad. El OuterCutOff es el cono donde la luz ya empieza a difuminarse hasta perder intensidad, este es de verdad el que da un poco de miedo. Lo primero que tenemos que hacer es discriminar en qué cono está nuestro punto. Para ello, sacamos el ángulo entre la dirección de la luz y el punto en el que incide.



Siendo Ld el vector unitario de a dónde apunta la luz y Lp el vector del fragmento a la luz. Al igual que en el 5.1.3, para sacar Lp simplemente tenemos que restar la posición del fragmento con la de la luz y normalizarlo.

Con este ángulo podemos discriminar en qué cono está. Básicamente es calcular si el ángulo que hemos calculado cae en el cono amarillo o en el naranja. Si cae en el amarillo, la intensidad es siempre la máxima. Si cae en el naranja, necesitamos saber qué tan lejos ha caído porque cuanto más lejos del amarillo más difuminado estará. Si se sale del cono naranja, no se ilumina.

Para esto, necesitamos calcular el ángulo límite, es decir, el ángulo a partir de donde empezamos ya a desvanecer la luz (osea, pasa del CutOff al OuterCutOff). Para eso simplemente restamos el valor de CutOff con el de OuterCutOff, ten en cuenta que estos valores los podrá modificar el usuario para decidir cómo difumina la luz su linterna. Ahora tenemos que interpolar nosotros a mano para ver, para ese ángulo que ahora ya sabemos que se va a difuminar, cuánto de difuminado está.



Interpolando sacamos el punto entre 1 y 0 en el que cae nuestro ángulo. Si es menor que 0, no nos importa porque cae fuera del cono naranja, si es mayor que 1 es que está dentro del cono amarillo.

5.4.4. Entidad Luz

Ahora que ya conocemos a nuestras tres fuentes de luz, tendremos que definirlas como entidades. Yo prefiero agrupar las tres en una sola entidad luz, aunque podrías separarlas si quieres. Para crear una nueva entidad luz sólo tenemos que hacer otro E_Light.hpp que herede de nuestra entidad genérica y actualizar el AllEnitities. Lo primero que vamos a hacer en esta nueva clase será un pequeño enumerado para guardar el tipo de luz.

```
// Los distintos tipos de luces soportadas por el motor
enum LightType {
    directional, pointLight, spotLight
};

// Clase luz
struct E_Light : virtual public E_Entity {
    ...

    // El tipo de luz
    LightType type{};
```

Esto nos servirá para internamente poder diferenciar qué tipos de luz estamos tratando. Vamos a guardar también los atributos comunes de todas las luces, la dirección para la direccional y la focal, la posición para la puntual y la focal... También guardaremos un puntero al shader que esté cargado al momento de dibujar.

```
// Un puntero al shader
Shader* shader{};
// Datos de las luces, la información es genérica, aunque
// solo se le darán uso a algunos dependiendo de la luz
glm::vec3 position{};
glm::vec3 direction{};
glm::vec3 ambient{};
glm::vec3 diffuse{};
glm::vec3 specular{};
// Constante, lineal y cuadrática, aprovecho un Vec 3
glm::vec3 attenuationValues{};
// El CutOff y el OuterCutOff, aprovecho un Vec 2
glm::vec2 spotAngles{};
```

Como cada luz requiere un tratamiento especial, creamos a parte de los constructores unos setters específicos para cada luz.

```
// Constructor por defecto por si solo quiere instanciar una luz
E_Light() = default;
// Constructor con un shader para inicializar el shader
E_Light(Shader* shader_) { shader = shader_; };

// Sete una luz direccional con el tipo
void setDirectionalLight(glm::vec3 dir, glm::vec3 amb,
  glm::vec3 dif, glm::vec3 spc) {
    type = LightType::directional;
    direction = dir;
    ambient = amb;
    diffuse = dif;
    specular = spc;
}
...
```

```
...
// Sete una luz puntual con el tipo
void setPointLight(glm::vec3 pos, glm::vec3 amb, glm::vec3 dif,
  glm::vec3 spc, glm::vec3 att) {
    type = LightType::pointLight;
    position = pos;
    ambient = amb;
    diffuse = dif;
    specular = spc;
    attenuationValues = att;
}

// Sete una luz focal con el tipo
void setSpotLight(glm::vec3 pos, glm::vec3 dir, glm::vec3 amb,
  glm::vec3 dif, glm::vec3 spc, glm::vec3 att, glm::vec2 cut) {
    type = LightType::spotLight;
    position = pos;
    direction = dir;
    ambient = amb;
    diffuse = dif;
    specular = spc;
    attenuationValues = att;
    spotAngles = cut;
}
```

Ahora normalmente crearíamos un método draw donde informamos los uniforms al shader, como con E_Model, y lo haremos, sin embargo, aún no hemos configurado nuestro Fragment Shader. Quizá es buen momento para ello.

5.5. COMPLICANDO EL FRAGMENT SHADER

Lo he retrasado todo lo que he podido, es hora de pasar todo lo que hemos aprendido a la práctica con nuestro Fragment Shader. Si te pierdes en algún punto te recomiendo volver a la parte correspondiente para volver a pillar el concepto.

Lo primero que vamos a hacer es recuperar esos tres objetos que pasamos como parámetro desde nuestro Vertex Shader.

```
in vec2 TexCoord;
in vec3 Normal;
in vec3 FragPos;
```

Ahora vamos a crear tres objetos dentro del shader para cada fuente de luz, así tendremos más claro qué propiedad pertenece a cada luz. He separado los componentes como la atenuación en floats distintos en vez de aprovechar el vector porque... bueno, en C++ quizá se vea claro pero ahora vamos a entrar en ecuaciones, cuanto más simple mejor.

```
// Objeto luz direccional
struct DirectionalLight{
    vec3 direction;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

// Objeto luz puntual
struct PointLight{
    vec3 position;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float constant;
    float linear;
    float cuadratic;
};

// Objeto luz focal
struct SpotLight{
    vec3 position;
    vec3 direction;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float constant;
    float linear;
    float cuadratic;
    float cutOff;
    float outerCutOff;
};
```

Vale, ya tenemos nuestras clases, vamos al main. Primero inicialicemos el color de nuestro fragmento, que siempre empezará en el 0, osea el negro, e irá cambiando conforme se vaya iluminando. También, aunque la normal que pasamos al Vertex Shader la teníamos normalizada, la que se habrá interpolado al llegar al Fragment Shader podría no estarlo, así que nos aseguramos. Por último, sacamos el vector director que apunta del fragmento a la cámara, que nos será útil en todas las luces.

```
uniform vec3 cameraPos;
...

// Creamos un color base para el color que ira
// aumentando según las luces
vec3 color = vec3(0.0f, 0.0f, 0.0f);
// Importantísimo normalizarla normal interpolada
vec3 normal = normalize(Normal);
// Sacamos el vector director que apunta desde el
// fragmento a la cámara
vec3 viewDir = normalize(cameraPos - FragPos);
```

Efectivamente, necesitamos la posición de la cámara en un unifrom, no es complicado de pasar. Antes de ponernos directamente con las luces, hay algo que tenemos que recordar. Estás haciendo un motor, no una escena, tu usuario

puede meter una luz o cincuenta. Por lo tanto, mientras implementamos las luces, tenemos que tener en mente que no podemos hacer un sistema estático sino uno dinámico, que responda a n luces de cualquier tipo. Aunque de momento, vamos a limitar que haya un máximo de diez luces de cada. También guardaremos unos uniform para que desde nuestro programa informemos al shader de exactamente cuántas hay.

```
// Valores dinámicos para saber la cantidad de luces
// que hay y de que tipo, por defecto están en 0.
uniform int n_directionalLights = 0;
uniform int n_pointLights = 0;
uniform int n_spotLights = 0;

// Definimos unos arrays de los distintos tipos de luces,
// como maximo admito 10 luces.
#define n_Max_directionalLights 10
uniform DirectionalLight dLight[n_Max_directionalLights];

#define n_Max_pointLights 10
uniform PointLight pLight[n_Max_pointLights];

#define n_Max_spotLights 10
uniform SpotLight sLight[n_Max_spotLights];
```

Ahora entramos a la salsa. Vamos a hacer tres métodos que llamaremos en el main, uno por cada luz, e iteraremos por estos arrays de luces que acabamos de crear para ir sumándole la intensidad de cada luz al color. Cada método, como cada fuente, tendrá parámetros distintos.

```
// Según el número de luces que haya añadido el usuario,
// se calculará tantas veces se necesite cada tipo ce luz
for(int i = 0; i < n_directionalLights; i++)
    color += calculateDirectionalLight(dLight[i],
      normal, viewDir);

for(int i = 0; i < n_pointLights; i++)
    color += calculatePointLight(pLight[i],
      normal, FragPos, viewDir);

for(int i = 0; i < n_spotLights; i++)
    color += calculateSpotLight(sLight[i],
      normal, FragPos, viewDir);
```

Los métodos pueden asustar al principio, pero no es más que la parte teórica que hemos dado hasta el momento aplicada. Te recomiendo darles una vuelta a los puntos 5.1 y 5.4. si ves algo que no te cuadra. Empecemos con las direccionales.

```
// Calculamos el efecto de las luces direccionales
// en el color del fragmento
vec3 calculateDirectionalLight(DirectionalLight dLight,
  vec3 Normal, vec3 viewDir){
    // Sacamos y normalizamos la dirección con la que el rayo
    // de luz colisiona con el fragmento, al revés
    vec3 lightDir = normalize(-dLight.direction);
    // Sacamos la intensidad entre 0 y el ángulo entre la
    // normal y la dirección de la luz
    float diffuseIntensity = max(dot(Normal, lightDir), 0.0);
    // Sacamos la dirección del reflejo con la inversa de la
    // luz de la dirección y la normal
    vec3 reflectDir = reflect(-lightDir, Normal);
    // Ahora calculamos la intensidad de esta luz según el
    // ángulo que hacen respecto a la cámara y
    // el brillo del material (de momento lo ponemos a 1.0)
    float specularIntensity = pow(max(
      dot(viewDir, reflectDir), 0.0), 1.0);

    // Recuperamos los componentes de Phong y le asignamos
    // su intensidad. La ambiental es global así que afecta en
    // todos los puntos con la misma intensidad
    vec3 ambient = dLight.ambient;
    vec3 diffuse = dLight.diffuse * diffuseIntensity;
    vec3 specular = dLight.specular * specularIntensity;
    // Devolvemos el resultado
    return (ambient + diffuse + specular);
}
```

Devolvemos un vector 3 con los valores de los tres componentes de la luz de Phong. Como viste arriba, estas intensidades se sumarán al color del fragmento. Ahora, vamos a la puntual y su atenuación.

```
vec3 calculatePointLight(PointLight pLight, vec3 Normal,
  vec3 FragPos, vec3 viewDir){
    // Para la puntual necesitamos saber la dirección
    // pero respecto al fragmento especifico
    vec3 lightDir = normalize(pLight.position - FragPos);
    // Difusa
    float diffuseIntensity = max(dot(Normal, lightDir), 0.0);
    // Especular
    vec3 reflectDir = reflect(-lightDir, Normal);
    // De nuevo, dejamos el brillo del material a 1.0 de momento
    float specularIntensity = pow(max(
      dot(viewDir, reflectDir), 0.0), 1.0);
    // Ahora calculamos la atenuación, primero sacando la
    // distancia entre la posición y el fragmento
    float distance = length(pLight.position - FragPos);
    // Aplicamos la fórmula de la atenuación usando la
    // constante, lineal y cuadrática
    float attenuation = 1.0 / (pLight.constant +
      (pLight.linear * distance) + (pLight.cuadratic *
      (distance * distance)));

    // Añadimos a los cálculos la atenuación
    vec3 ambient = pLight.ambient * attenuation;
    vec3 diffuse = pLight.diffuse * diffuseIntensity *
      attenuation;
    vec3 specular = pLight.specular * specularIntensity *
      attenuation;
    return (ambient + diffuse + specular);
```

Lo complicado aquí es la atenuación, pero como ves, aunque en código queda un poco más lioso, es sólo aplicar la fórmula. Las luces focales son un poco más especiales, si has entendido bien lo del CutOff y el OuterCutOff entonces igual no lo es tanto.

```
vec3 calculateSpotLight(SpotLight sLight, vec3 normal,
  vec3 FragPos, vec3 viewDir){
    vec3 lightDir = normalize(sLight.position - FragPos);
    // Difusa
    float diffuseIntensity = max(dot(normal, lightDir), 0.0);
    // Especular
    vec3 reflectDir = reflect(-lightDir, normal);
    // Vuelvo a repetir lo de que de momento el brillo del
    // material a 1.0 para que no se nos olvide
    float specularIntensity = pow(max(
      dot(viewDir, reflectDir), 0.0), 1.0);
    // Atenuacion
    float distance = length(sLight.position - FragPos);
    float attenuation = 1.0 / (sLight.constant +
      (sLight.linear * distance) + (sLight.cuadratic *
      (distance * distance)));
    // Calculamos los ángulos de cierre para la luz dirigida,
    // para saber si entran dentro o no de un cono y de cuál.
    // Usamos la dirección de la luz (al revés) y el vector
    // director del fragmento a la luz
    float limitAngle = dot(lightDir,
      normalize(-sLight.direction));
    // Primero calculamos el ángulo limite que
    //separa CutOff de OuterCutOff
    float fadeOffAngle = sLight.cutOff - sLight.outerCutOff;
    // Hacemos la interpolación de intensidades. Devolverá un
    // número más cercano a 1.0 si está más cerca del CutOff y
    // más cercano a 0,0 si lo está del OuterCutOff. Clampeamos
    // el valor para que si está fuera de ambos conos sea un
    // 0,0 y si está completamente dentro del CutOff sea 1.0.
    float intensity = clamp((limitAngle - sLight.outerCutOff)
       / fadeOffAngle, 0.0, 1.0);
    // Aplicamos todos los valores calculados,
    //añadiendo la intensidad interpolada
    vec3 ambient = sLight.ambient * attenuation * intensity;
    vec3 diffuse = sLight.diffuse * diffuseIntensity *
       attenuation *  intensity;
    vec3 specular = sLight.specular * specularIntensity *
       attenuation * intensity;
    return (ambient + diffuse + specular);
}
```

Vale, todo listo. Como ves no hay nada nuevo que no haya explicado ya. Lo último que hará nuestro Fragment Shader es, en el main, una vez hayamos iterado por todas las luces y habiendo sumado estas intensidades al color, devolverlo como el color del fragmento.

```
// Una vez hemos sacado las luces, se pasa el resultado total
FragColor = vec4(color, 1.0);
```

Y ya estaría, nuestro Fragment Shader con sistema de luces dinámico y tope. Espero que no te haya liado mucho. De momento no te paso el código porque, sorpresa, volveremos a este shader en el futuro para las texturas. De momento, vamos a ver cómo dibujaríamos nuestra entidad luz con este nuevo Fragment Shader.

5.6. DIBUJANDO LAS LUCES

Volvamos a nuestra E_Light. Para dibujar las luces tenemos que tener en cuenta los arrays dinámicos del shader. Normalmente metemos un uniform en plan "variable = 1", ahora tendremos que meterlos más en plan "objeto[idDelArray].variable = 1". Para eso necesitaremos ir teniendo un control de cuántas luces de cada tipo se han ido creando y guardar ese contador dentro de la entidad.

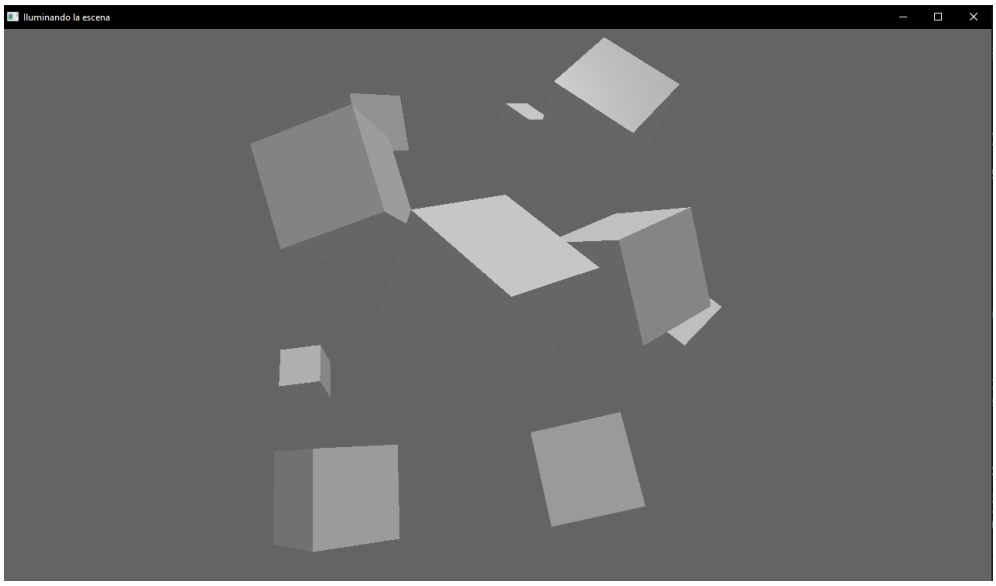
Como para enviar la información al shader usamos los nombres de los uniform, para un sistema dinámico de luces también necesitaremos un sistema dinámico de nombres. Vamos a crear dos cadenas, una para el nombre de la luz y otra para la id de la misma. De forma que la primera vez que se cree una luz direccional le asignemos a la del nombre "dLight[", le sumemos la id "0" y luego cerramos con el nombre del uniform "].ambient". Así, cuando metamos una segunda luz direccional lo único que cambiará es que la id será un 1 y si metemos otro tipo de luz lo que cambia es el nombre. Dinámico.

```
// Método de dibujado, en verdad no necesita ni la
// matriz acumulada ni la cámara en este caso
void draw(glm::mat4 matrix = glm::mat4(1.0f),
  E_Camera* principalCamera = nullptr) override {
    // Si hay un shader, hace el dibujado si no pues pa que
    if (shader) {
        // Nos aseguramos que se use el shader
        shader->use();
        // Preparamos las cadenas para el sistema dinámico
        std::stringstream confg{};
        std::string light{};
        // Segun el tipo, manda una información al shader u otra.
        switch (type) {
        case directional:
            confg.str("");
            light = "dLight[";
            confg << light << id << "].direction";
            shader->setVec3(confg.str(),
              glm::vec3(direction.x, direction.y, direction.z));
            // Reseteamos la cadena dinámica
            confg.str("");
            confg << light << id << "].ambient";
            shader->setVec3(confg.str(),
              ambient.x, ambient.y, ambient.z);
            ...
```

```
            ...
            confg.str("");
            confg << light << id << "].diffuse";
            shader->setVec3(confg.str(),
              diffuse.x, diffuse.y, diffuse.z);
            confg.str("");
            confg << light << id << "].specular";
            shader->setVec3(confg.str(),
              specular.x, specular.y, specular.z);
            break;
        case pointLight:
            confg.str("");
            light = "pLight[";
            confg << light << id << "].position";
            shader->setVec3(confg.str(),
              glm::vec3(position.x, position.y, position.z));
            confg.str("");
            confg << light << id << "].ambient";
            shader->setVec3(confg.str(),
              ambient.x, ambient.y, ambient.z);
            confg.str("");
            confg << light << id << "].diffuse";
            shader->setVec3(confg.str(),
              diffuse.x, diffuse.y, diffuse.z);
            confg.str("");
            confg << light << id << "].specular";
            shader->setVec3(confg.str(),
              specular.x, specular.y, specular.z);
            confg.str("");
            confg << light << id << "].constant";
            shader->setFloat(confg.str(), attenuationValues.x);
            confg.str("");
            confg << light << id << "].linear";
            shader->setFloat(confg.str(), attenuationValues.y);
            confg.str("");
            confg << light << id << "].cuadratic";
            shader->setFloat(confg.str(), attenuationValues.z);
            break;
        case spotLight:
            confg.str("");
            light = "sLight[";
            confg << light << id << "].position";
            shader->setVec3(confg.str(),
              glm::vec3(position.x, position.y, position.z));
            confg.str("");
            confg << light << id << "].direction";
            shader->setVec3(confg.str(),
              glm::vec3(direction.x, direction.y, direction.z));
            confg.str("");
            confg << light << id << "].ambient";
            shader->setVec3(confg.str(),
              ambient.x, ambient.y, ambient.z);
            confg.str("");
            confg << light << id << "].diffuse";
            shader->setVec3(confg.str(),
              diffuse.x, diffuse.y, diffuse.z);
            confg.str("");
            confg << light << id << "].specular";
            shader->setVec3(confg.str(),
              specular.x, specular.y, specular.z);
            ...
```

```
            ...
            confg.str("");
            confg << light << id << "].constant";
            shader->setFloat(confg.str(), attenuationValues.x);
            confg.str("");
            confg << light << id << "].linear";
            shader->setFloat(confg.str(), attenuationValues.y);
            confg.str("");
            confg << light << id << "].cuadratic";
            shader->setFloat(confg.str(), attenuationValues.z);
            confg.str("");
            confg << light << id << "].cutOff";
            shader->setFloat(confg.str(), spotAngles.x);
            confg.str("");
            confg << light << id << "].outerCutOff";
            shader->setFloat(confg.str(), spotAngles.y);
            break;
        }
    }
};
```

Como ves la idea es sencilla, pero para repetirlo por cada uniform se hace tedioso. Si se te ocurre a ti una mejor solución te animo a intentarla. Con esto, nuestro Fragment Shader tiene toda la información necesaria de las normales desde nuestro Vertex Shader y de las luces desde nuestra E_Light como para hacer todos los cálculos que necesita e iluminar la escena. Sin embargo, si teóricamente creases ahora la luz y ejecutases nuestro sistema verías algo parecido a lo siguiente:



¿A dónde se ha ido todo el color? Bueno, es que hemos calculado intensidades, qué tanto brilla o no ese fragmento del cubo, y el brillo como tal es una escala de grises. Necesitamos tener en cuenta el color del cubo y sobre ese color añadirle el brillo. Pero me parece un poco aburrido añadirle sólo un amarillo plano, vamos a meterles texturas al fin.

5.7. TEXTURIZANDO EL CUBO

Como comenté en el capítulo 5.2, una textura es una imagen 2D y para añadirle textura al cubo sólo teneos que decirles a sus vértices qué coordenada de la imagen le corresponden. Como te imaginarás, para poder hacer eso primero necesitaremos cargar la imagen 2D, para eso vamos a apoyarnos en una librería llamada STB.

La cabecera de STB se puede descargar desde su repositorio de GitHub[23]. Si tienes bien configurados los entornos como hicimos en el capítulo 2, pues con arrastrar dicha cabecera a tu carpeta de include/librerías debería ser suficiente para que tu aplicación la reconozca, si no releyendo el capítulo refrescarás cómo hacerlo.

Las texturas son un recurso, yo personalmente como siempre van a estar enlazadas a algún modelo no veo necesario crear un recurso nuevo como tal, pero sí que tenemos que hacer que nuestro gestor de recursos maneje la lectura y almacenamiento de las imágenes, concretamente el recurso modelo. Antes de nada, vamos a actualizar un poco nuestro recurso malla para que pueda contener las texturas con una nueva clase.

```
// Clase textura
struct Texture {
    unsigned int id{};
    std::string type{};
    std::string path{};
};

...

// El Constructor ahora recibe también texturas
R_Mesh(std::vector<Vertex> vertices, std::vector<unsigned int>
indices, std::vector<Texture> textures){
    this->vertices = vertices;
    this->indices = indices;
    this->textures = textures;
    // Cuando tenemos los datos los procesamos
    setupMesh();
};

...

// Los vectores con la información almacenada, también le
añadimos una de texturas
std::vector<Vertex> vertices{};
std::vector<unsigned int> indices{};
std::vector<Texture> textures{};
```

[23] Cabecea STB

La clase textura es sencilla, guardamos el id en memoria de OpenGL (similar a cómo guardábamos la id de los buffers), el tipo de textura y la ruta de la imagen. Por lo demás simplemente añadimos la información al flujo de la clase. Como el Vertex Shader no necesita saber nada de las texturas, no hace falta que toquemos los buffers, podemos centrarnos en cargar las texturas con nuestro gestor de recursos.

Primero, vamos a crear un vector dentro de ResourceManager.hpp para guardar las texturas ya leídas y evitar recargar texturas que ya hemos cargado.

```
// Incluimos en nuestro ResourceManager STB
#include <stb_image.h>

...

// Vector que almacena las texturas cargadas
std::vector<Texture> texturesLoaded{};
```

Ahora declaramos el método de carga de texturas en el R_Model.hpp. Como he dicho antes, una textura irá siempre con un modelo por lo que veo adecuado que sea este quien lo cargue. Recibirá la ruta del archivo de la imagen, el tipo de textura y una referencia al vector de texturas cargadas.

```
// Anade una textura a todas las mallas del modelo
void addTexture(const std::string& path, const std::string&
       type, std::vector<Texture>& texturesLoaded);
```

La implementación la tendremos que hacer ya en nuestro ResourceManager.cpp, hay que mencionar que STB es un poco especialita y necesita compilarse, así que también la importaremos allí. Otra cosa que tenemos que hacer con STB es indicarle que necesitamos importar la parte del tratamiento de imágenes, que si no se la deja sin importar. El proceso para cargar una imagen es parecido a cómo configurábamos un buffer con OpenGL.

```
// Implementacion de addTexture para R_Model
void R_Model::addTexture(const std::string& path, const
  std::string& type, std::vector<Texture>& texturesLoaded) {
    // Comprobamos si la textura ya esta cargada
    for (auto& t : texturesLoaded) {
        if (t.path == path) {
            for (auto& mesh : meshes) {
                mesh.textures.push_back(t);
            }
            // Ponemos el mixValue a 0 para sólo esté la textura
            mixValue = 0.0f;
            return;
        }
    }
    ...
```

```
    // No existe, la cargamos, Primero generamos una id
    unsigned int textureID{};
    // Le decimos a OpenGL que genere el espacio para la textura
    glGenTextures(1, &textureID);
    // La altura, la anchura y los componentes de la imagen
    int width{}, height{}, nrComponents{};
    // Cargamos la imagen con STB
    unsigned char* data = stbi_load(
      path.c_str(), &width, &height, &nrComponents, 0);
    // Si la carga ha sido exitosa, entonces continuamos
    if (data) {
        // Miramos el formato de la imagen según sus componentes
        GLenum format{};
        // Un componente, será sólo roja la imagen
        if (nrComponents == 1) format = GL_RED;
        // Con tres roja, verde y azul (RGB)
        else if (nrComponents == 3) format = GL_RGB;
        // Y si tiene canal alpha para transparencias, cuatro
        else if (nrComponents == 4) format = GL_RGBA;

        // Enlazamos la id que generamos antes
        glBindTexture(GL_TEXTURE_2D, textureID);
        // Le decimos a OpenGL que cargue la textura
        glTexImage2D(
            GL_TEXTURE_2D,    // El tipo de textura objetivo
            0,                // Nivel de imagen, por defecto es 0
            format,           // Formato de la imagen
            width,            // Ancho
            height,           // Alto
            0,                // El borde de la imagen
            format,           // Formato de los píxeles,
                              // podría ser diferente
            GL_UNSIGNED_BYTE, // El tipo de datos de los píxeles
            data              // Puntero a la imagen cargada
        );
        stbi_image_free(data);
    } else {
        // Si no ha cargado, debugeamos
        std::cout <<
            "Texture failed to load: " << path << std::endl;
    }

    // Generamos un objeto textura y le asignamos los valores
    // recuperados, lo metemos en el array de texturas cargadas.
    Texture texture{};
    texture.id = textureID;
    texture.type = type;
    texture.path = path;
    texturesLoaded.push_back(texture);

    // Actualizamos las mallas del modelo con la nueva textura
    for (auto& mesh : meshes) {
        mesh.textures.push_back(texture);
    }
    // Ponemos el mixValue a 0 para que se vea sólo la textura
    mixValue = 0.0f;
}
```

Igual te estás preguntando “¿Ey, qué es eso de mixValue?”. Bueno, espero que sólo te estés preguntando eso, significaría que el resto lo he explicado bien jeje. El mixValue será simplemente un valor que usaremos para saber cuándo pintar

un modelo por su color y cuándo por su textura. Imagina un cubo amarillo con una textura de una caja de madera. Cuando este valor sea 0, verás completamente la caja, cuando sea 1 verás sólo un cubo amarillo y cuando esté a 0.5 verás una mezcla de ambas difuminadas. Este atributo lo definiremos en R_Model.hpp, por defecto que muestre el color del modelo:

```
float mixValue{ 1.0f };
```

Ahora nos aseguramos de que el E_Model, al dibujarse, se lo pase también al Fragment Shader en un uniform.

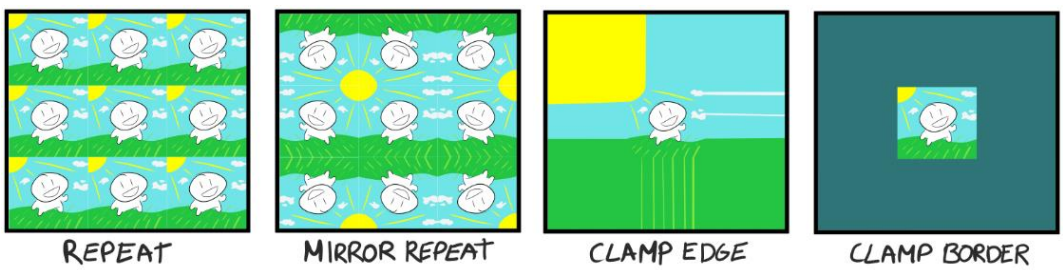
```
// El mixValue del modelo
actualShader.setFloat("mixValue", model->getMixValue());
```

Ya nos preocuparemos por este uniform en el futuro. Volvamos a la textura, que ya la tenemos cargada en nuestra malla. Sin embargo, no le hemos dicho a OpenGL cómo deberá leer luego cada pixel de la imagen, simplemente se la hemos cargado y ya. Esto nos quita control sobre qué hará OpenGL, es nuestra responsabilidad decirle a OpenGL cómo tratar cada texel (píxel de la textura). A esto se le conoce como el muestreo de las texturas o texture sampling.

5.7.1. Wrapping

Lo primero que tenemos que decirle a OpenGL es qué pasará cuando un modelo se salga del rango de coordenadas. Es decir, que intente acceder a una coordenada que se salga del [0, 1]. Existen cuatro, dependiendo del caso te puede interesar una u otra.

- GL_REPEAT: Se repite los bordes la textura tal cual, es la default.
- GL_MIRRORED_REPEAT: Igual, solo que refleja la imagen cada vez.
- GL_CLAMP_TO_EDGE: Fija las coordenadas al límite, dando la sensación de estirar la imagen.
- GL_CLAMP_TO_BORDER: Las coordenadas que se salgan se interpretarán como un borde. Necesitarás definir el color del mismo.



El wrapping se aplica a cada coordenada de la textura. Para distinguirlas de las de las XYZ, se les llama S y T. Ahora tenemos que indicarle a OpenGL cuál queremos, podemos añadirlo en ResourceManager.cpp antes de liberar la imagen de la memoria.

```
// Le metemos parámetros a la textura 2D, en ambas coordenadas
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

5.7.2. Filtering

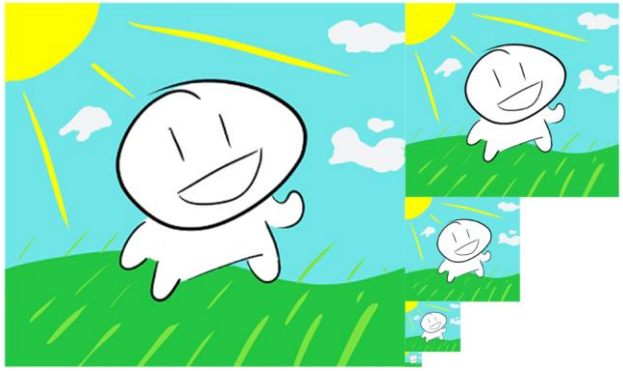
Las coordenadas de las texturas no dependen de la resolución, esto hace que si tienes un objeto grande y una textura de baja calidad varias coordenadas de tu objeto apunten al mismo punto de la textura, pudiendo generar patrones extraños. Para evitar esto, se usa un filtro de suavizado de las texturas. Hay varios, pero principalmente se usan dos:

- GL_NEAREST: Escoge el texel que más se acerque al centro, es la que viene por defecto.
- GL_LINEAR: Interpola el valor del texel con el de sus texel vecinos, produce un resultado más suave, pero requiere más cálculo.

Podemos configurar el parámetro del filtering justo debajo de los del wrapping, pero antes de decirte cómo te voy a hablar del mipmapping.

5.7.3. Mipmaps

Imagina que tienes un objeto con una textura, como lo tienes cerca lo ves perfecto. Ahora imaginas que te mueves muy muy lejos de ese objeto, pero donde aún lo puedes ver. No sólo va a tener el mismo problema que teníamos con el filtering de la resolución (aunque al revés), sino que encima vamos a tener la textura grande ahí en memoria pa na, porque el objeto lo vas a ver pequeño. Hay muchas soluciones, pero la estándar son los minmaps que no son más que un conjunto de versiones de distintos tamaños de la misma textura.



Si se lo indicas, OpenGL decidirá qué mipmap es el más adecuado para pintar la textura. Es decir, si estás muy cerca cogerá la textura más grande y conforme te vayas alejando irá cogiendo las más pequeñas. Como todo va por coordenadas, el usuario no notará el cambio.

Al igual que el filtering, puedes configurar cómo va a ser el salto entre mipmaps, el NEAREST para que el texel de la nueva resolución se quede con el que tenga el color más cercano al centro y el LINEAR para que haga una interpolación de sus vecinos. Hay que tener en cuenta que tanto el filtering como el mipmaping se configuran diferente dependiendo de si nos alejamos (Minifying) que si nos acercamos (Magnifying). Ahora sí, añadimos estos parámetros a las texturas justo debajo de las del wrapping.

```
// Para cuando nos alejamos, usamos el Linear pero con Mipmaps
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
      GL_LINEAR_MIPMAP_LINEAR);
// Para cuando nos acercamos, usamos el Linear general
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
      GL_LINEAR);
// Generamos los mipmaps de la textura cargada
glGenerateMipmap(GL_TEXTURE_2D);
```

5.7.4. Dibujando las texturas

Vale, vale, tenemos las texturas cargadas y listas en nuestras entidades. ¿Cómo las dibujamos ahora? Bien, vamos a modificar un poco nuestro Fragment Shader y nuestro recurso malla. Lo primero que vamos a hacer en el shader es definir una clase material. Al fin y al cabo, textura o no, todo objeto debería tener unas propiedades que se asocian a su material.

```
// Objeto material, con todo lo necesario
// para calcular el color de los materiales
struct Material{
    // Textura del color
    sampler2D diffuse;
    // Textura de los brillos
    sampler2D specular;
    float shininess;
    vec3 color;
};
```

Aquí hay varias cosas que diseccionar. Lo primero es el sampler2D, no es más que el tipo para las texturas en OpenGL. Luego, hay dos, una para el color y otra para los brillos. No voy a entrar mucho en detalle, pero se puede usar una segunda textura en escala de gris para marcar las zonas brillantes de un objeto. Piensa en una textura que marque en una placa de metal una zona que ha sido rallada y, por lo tanto, ya no debería de brillar. Para eso se usan.

Guardamos un uniform tanto del mixValue que vimos antes como de una instancia del material.

```
uniform Material material;
uniform float mixValue;
```

Para poder calcular el color del fragmento según su coordenada en la textura, OpenGL nos lo hace automáticamente con la función textura. Recuerda que en el VertexShader ya nos pasamos la coordenada de la textura al FragmentShader.

```
texture(material.diffuse, TexCoord)
```

Ahora que tenemos nuestro material y su textura, vamos a hacer un mix del color de la textura con el color del material en sí, usando el mixValue, Esto es lo que hará que si está a 0.5 por ejemplo salga un combinado de ambas. Ten en cuenta que para la luz especular usaremos la textura especular, no la del color.

```
vec3 calculateDirectionalLight(DirectionalLight dLight,
  vec3 Normal, vec3 viewDir){
    ...

    // Multiplicamos las intensidades por los valores de
    // las texturas mezcladas con el color según el mixValue
    vec3 ambient = dLight.ambient *
      mix(vec3(texture(material.diffuse, TexCoord)),
      material.color, mixValue);
    vec3 diffuse = dLight.diffuse * diffuseIntensity *
      mix(vec3(texture(material.diffuse, TexCoord)),
      material.color, mixValue);
    // Para la intensidad especular, la textura especular
    vec3 specular = dLight.specular * specularIntensity *
      mix(vec3(texture(material.specular, TexCoord)),
      material.color, mixValue);
    // Devolvemos el resultado
    return (ambient + diffuse + specular);
}

vec3 calculatePointLight(PointLight pLight, vec3 Normal,
  vec3 FragPos, vec3 viewDir){
    ...

    // Mismo concepto, pero nos aseguramos de que la atenuación
    // se aplique al color ya calculado de la textura, no antes
    vec3 ambient = (pLight.ambient * mix(vec3(texture(
      material.diffuse, TexCoord)), material.color, mixValue)) *
      attenuation;
    vec3 diffuse = (pLight.diffuse * diffuseIntensity *
      mix(vec3(texture(material.diffuse, TexCoord)),
      material.color, mixValue)) * attenuation;
    vec3 specular = (pLight.specular * specularIntensity *
      mix(vec3(texture(material.specular, TexCoord)),
      material.color, mixValue)) * attenuation;
    return (ambient + diffuse + specular);
}
...
```

```
...
vec3 calculateSpotLight(SpotLight sLight, vec3 normal,
  vec3 FragPos, vec3 viewDir){
    ...

    // Igual que antes, nos aseguramos de que la atenuación y
    // el difuminado de la intensidad se aplique después
    vec3 ambient = (sLight.ambient * mix(vec3(texture(
      material.diffuse, TexCoord)), material.color, mixValue)) *
      attenuation * intensity;
    vec3 diffuse = (sLight.diffuse * diffuseIntensity *
      mix(vec3(texture(material.diffuse, TexCoord)),
      material.color, mixValue)) * attenuation *  intensity;
    vec3 specular = (sLight.specular * specularIntensity *
      mix(vec3(texture(material.specular, TexCoord)),
      material.color, mixValue)) * attenuation * intensity;
    return (ambient + diffuse + specular);
}
```

Antes de avanzar, recuerda cambiar en todos los cálculos de la specularIntensity el 1.0 placeholder que metimos por nuestro nuevo y reluciente material.shininess. Ahora sí ya sólo necesitas informar estos uniform desde el recurso malla y la entidad modelo a la hora de dibujar. Igual que con las luces, vamos a hacer un sistema dinámico el tipo de la clase Texture para saber si es una textura de color, difusa, o una de brillo, especular. Primero vemos el recurso malla.

```
// Metodo de dibujado en OpenGL
void draw(Shader& shader){
    /// Inicializamos el contador para las texturas
    uint8_t texturas = 0;
    // Busca texturas difusas y especulares en sus texturas
    for (const auto& tex : textures) {
        if (tex.type == "textureDiffuse") {
            // Activamos la textura por el número en el que vaya
            glActiveTexture(GL_TEXTURE0 + texturas);
            // Enlazamos la ID de cuando la cargamos
            glBindTexture(GL_TEXTURE_2D, tex.id);
            // La pasamos como uniform
            shader.setInt("material.diffuse", texturas);
            texturas++;
        }
        else if (tex.type == "textureSpecular") {
            glActiveTexture(GL_TEXTURE0 + texturas);
            glBindTexture(GL_TEXTURE_2D, tex.id);
            shader.setInt("material.specular", texturas);
            texturas++;
        }
    }
    // Activamos la textura inicial, la 0, por si no tenía textura
    glActiveTexture(GL_TEXTURE0);
    // Anidamos el Vertex Array Object
    glBindVertexArray(VAO);
    // Dibujamos los triangulos segun los indices
    glDrawElements(GL_TRIANGLES,
      static_cast<int>(indices.size()), GL_UNSIGNED_INT, 0);
    // Desanidamos el Vertex Aray Object
    glBindVertexArray(0);
};
```

Después modificamos la entidad modelo. Aprovechamos y metemos también el uniform de la posición de la cámara.

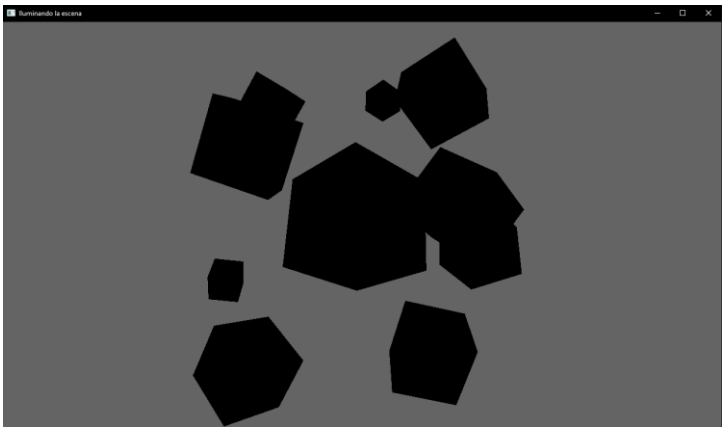
```
// Le pasamos la posición de la cámara
actualShader.setVec3("cameraPos",
      glm::vec3(principalCamera->position.x,
      principalCamera->position.y,
      principalCamera->position.z));
// Le pasamos un material base para el color, si ya hay una textura
// el mixValue será 0 por lo que este material no hará nada,
// pero es útil cuando no hay texturas en el modelo
actualShader.setInt("material.diffuse", 0);
// De base todo brilla, así lo hará aunque no haya textura de brillo
actualShader.setInt("material.specular", 1);
// De momento le metemos un brillo base
actualShader.setFloat("material.shininess", 64.0f);
// El color definido por el usuario, de base es el amarillo
actualShader.setVec3("material.color",
      glm::vec3(color.r, color.g, color.b));
```

Con esto, ya estamos listos para modificar la escena con nuestras luces y nuestras texturas. Ha sido un viaje, pero cuando lo tengamos terminado habrá valido la pena, créeme.

Igual no ves necesaria toda esta parafernalia de la textura de brillos, sistemas dinámicos, mixValue, etc. Bueno, recuerda que nuestra idea es darle al usuario más opciones. Para un motor gráfico, cuanto más podamos hacer mejor. Aun así, por no alargarme aún más, he dejado cosas en el tintero. Por ejemplo, si te fijas nuestro motor tiene el muestreo de texturas completamente estático, el usuario no podría cambiarlo. Ese tipo de cosas podrías mejorarlas si quieres ampliar el motor.

5.8. UNA ESCENA CUBISTA, ORDENADA E ILUMINADA

Ya tenemos todo listo para mejorar nuestra escena. De hecho, si ejecutas ahora el programa, verás que todos los modelos están completamente negros.



Normal, como no hemos definido nuestras luces todas las intensidades están a 0, el negro. Vamos a definir los métodos de creación de nuestras luces. Será muy parecido a los métodos de la creación de nuestros cubos y cámaras.

```
// Los métodos para crear las luces
Node* createDirectionalLight(glm::vec3 dir, glm::vec3 amb,
  glm::vec3 dif, glm::vec3 spc, std::string shader) {
    auto nodeLight = std::make_unique<Node>();
    nodes.push_back(std::move(nodeLight));
    auto& nLight = nodes[nodes.size() - 1];
    auto entityLight = std::make_unique<E_Light>();
    lights.push_back(std::move(entityLight));
    auto& eLight = lights[lights.size() - 1];
    // Como ya estará cargado en memoria el gestor de
    // recurso nos devolverá su ID de OpenGL
    eLight->shader = &RM.getShader(
      shader, shader, shader)->shader;
    // Crea una luz direccional en la entidad luz con los parámetros
    eLight->setDirectionalLight(dir, amb, dif, spc);
    // Marcamos la id como el numero actual de luces direccionales
    eLight->id = static_cast<int>(numLights.x);
    nLight->setEntity(std::move(eLight.get()));
    sceneRoot->addChild(std::move(nLight.get()));
    // Aumenta el contador de luces direccionales
    numLights.x += 1;
    return sceneRoot->getChildren()
      [sceneRoot->getChildren().size() - 1];
}

Node* createPointLight(glm::vec3 pos, glm::vec3 amb, glm::vec3 dif,
  glm::vec3 spc, glm::vec3 att, std::string shader) {
    auto nodeLight = std::make_unique<Node>();
    nodes.push_back(std::move(nodeLight));
    auto& nLight = nodes[nodes.size() - 1];
    auto entityLight = std::make_unique<E_Light>();
    lights.push_back(std::move(entityLight));
    auto& eLight = lights[lights.size() - 1];
    eLight->shader = basicShader;
    // Crea una luz puntual en la entidad luz con los parámetros
    eLight->setPointLight(pos, amb, dif, spc, att);
    // Esta vez usamos el número de luces puntuales
    eLight->id = static_cast<int>(numLights.y);
    nLight->setEntity(std::move(eLight.get()));
    sceneRoot->addChild(std::move(nLight.get()));
    // También aumentamos el número de luces puntuales
    numLights.y += 1;
    return sceneRoot->getChildren()
      [sceneRoot->getChildren().size() - 1];
}
...
```

```
...

Node* createSpotLight(glm::vec3 pos, glm::vec3 dir, glm::vec3 amb,
  glm::vec3 dif, glm::vec3 spc, glm::vec3 att, glm::vec2 cut,
  std::string shader) {
    auto nodeLight = std::make_unique<Node>();
    nodes.push_back(std::move(nodeLight));
    auto& nLight = nodes[nodes.size() - 1];
    auto entityLight = std::make_unique<E_Light>();
    lights.push_back(std::move(entityLight));
    auto& eLight = lights[lights.size() - 1];
    eLight->shader = basicShader;
    // Crea una luz focal en la entidad luz con los parámetros
    eLight->setSpotLight(pos, dir, amb, dif, spc, att, cut);
    // Esta vez aumentamos el número de luces focales
    eLight->id = static_cast<int>(numLights.z);
    nLight->setEntity(std::move(eLight.get()));
    sceneRoot->addChild(std::move(nLight.get()));
    // De nuevo, aumentamos el número de luces focales
    numLights.z += 1;
    return sceneRoot->getChildren()
      [sceneRoot->getChildren().size() - 1];
}
```

Con estos métodos, en el main después de cargar el shader, ya podemos crear nuestras primeras luces de cada tipo. ¡Qué emoción! Crearemos una luz direccional muy suavita, para que haya una luz global tenue, una puntual arriba justo del cubo amarillo y una focal que usaremos de linterna para la cámara. De esta última tendremos que ir actualizando su posición en el bucle de renderizado así que recuperamos el nodo.

```
// Creamos las luces, le pasamos el mismo nombre del shader
createDirectionalLight(
    glm::vec3(-0.2f, -1.0f, -0.3f), glm::vec3(0.1f, 0.1f, 0.1f),
    glm::vec3(0.1f, 0.1f, 0.1f), glm::vec3(0.2f, 0.2f, 0.2f),
    "base");
createPointLight(
    glm::vec3(-3.8f, 0.0f, -12.3f), glm::vec3(0.2f, 0.2f, 0.2f),
    glm::vec3(0.8f, 0.8f, 0.8f), glm::vec3(1.0f, 1.0f, 1.0f),
    glm::vec3(1.0f, 0.09f, 0.032f), "base");
auto* spotLight = createSpotLight(
    cameras.at(0)->position, cameras.at(0)->front,
    glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(1.0f, 1.0f, 1.0f),
    glm::vec3(1.0f, 1.0f, 1.0f), glm::vec3(1.0f, 0.09f, 0.032f),
    glm::vec2(glm::cos(glm::radians(12.5f)),
    glm::cos(glm::radians(15.0f))), "base");
```

Tomate un segundo para entender el por qué de cada parámetro. Si has entendido realmente la teoría, no debería ser muy complicado saber el por qué de cada cosa. Quizá lo que sí te cueste entender es cómo impacta realmente poner un 0.3 o un 0.6 en alguno de los valores. Cuando terminemos la escena, te recomiendo encarecidamente que juegues con estos valores, experimentes, es la mejor manera de aprender de verdad cuánto impacto tienen en la escena.

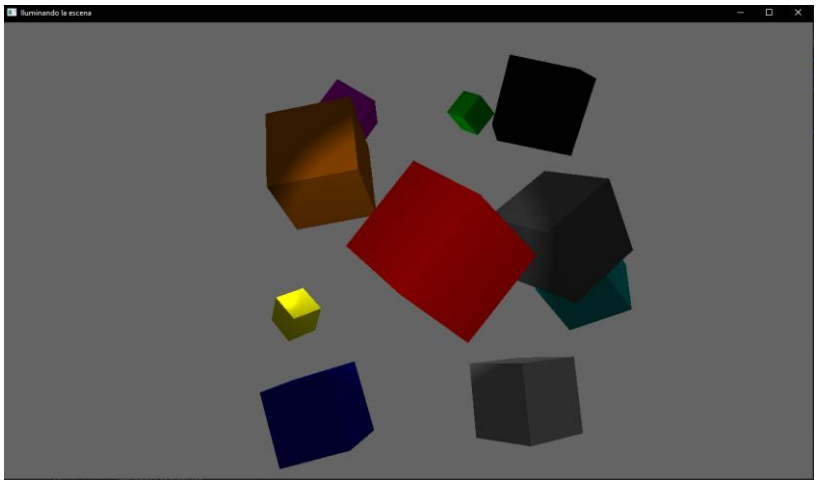
Ahora hay que tener en cuenta que tenemos que avisarle al shader de cuántas luces ha creado el usuario. Esto lo haremos dentro del bucle de renderizado. También, ten en cuenta que, si dibujas primero un cubo y después la luz, el cubo que se haya dibujado antes no se verá iluminado. Por eso, antes de recorrer el árbol, iteramos por nuestro vector de luces y las dibujamos.

```
// Renderizamos primero las luces, así da igual el orden en los
// que las haya metido el usuario en el árbol
for (auto& l : lights)
    l->draw(glm::mat4{ 1.0f }, sceneRoot->getPrincipalCamera());
// Le decimos al shader cuantas luces hay de cada tipo
basicShader->setInt("n_directionalLights",
       static_cast<int>(numLights.x));
basicShader->setInt("n_pointLights", static_cast<int>(numLights.y));
basicShader->setInt("n_spotLights", static_cast<int>(numLights.z));
```

Lo único que nos queda antes de meternos con las texturas, actualizar en el bucle de renderizado, y antes de dibujar, la posición de la luz focal para que alumbre desde la cámara.

```
// Casteamos la entidad genérica a entidad luz
auto* sp = dynamic_cast<E_Light*>(spotLight->getEntity());

// Actualizamos la posición y dirección de la luz con la de la cámara
sp->position = cameras.at(0)->position;
sp->direction = cameras.at(0)->front;
```

Tu escena ahora debería de verse mucho más bonita.



Si te fijas, todos los cubos están medianamente iluminados, claramente puedes ver el foco de tu linterna iluminando los cubos cerca tuya conforme te muevas y el cubo amarillo siempre aparece iluminado por arriba (nuestra luz puntual). ¡Y se hizo la luz! Es hora de que se hagan también las texturas.

Te voy a pasar una textura de una caja[24], con ella y nuestro método en la entidad modelo vamos a asignarle sólo al cubo rojo una textura. Primero hacemos un método en nuestra E_Modelo.hpp que use la función de addTexture ya implementada,

```
// Busca y asigna una textura al modelo
void setTexture(std::string path) { if (model)
      model->addTexture(path, "textureDiffuse",
      RM->texturesLoaded); };
```

Después llamamos a nuestra función en el main, después de ya haber creado los cubos.

```
// Cargamos la textura para el cubo rojo, el primero
dynamic_cast<E_Model*>(
      cubo1->getEntity())->setTexture("./Resources/Caja.png");
```

¡Ahora carga la escena y...! Un segundo, todos son cajas.

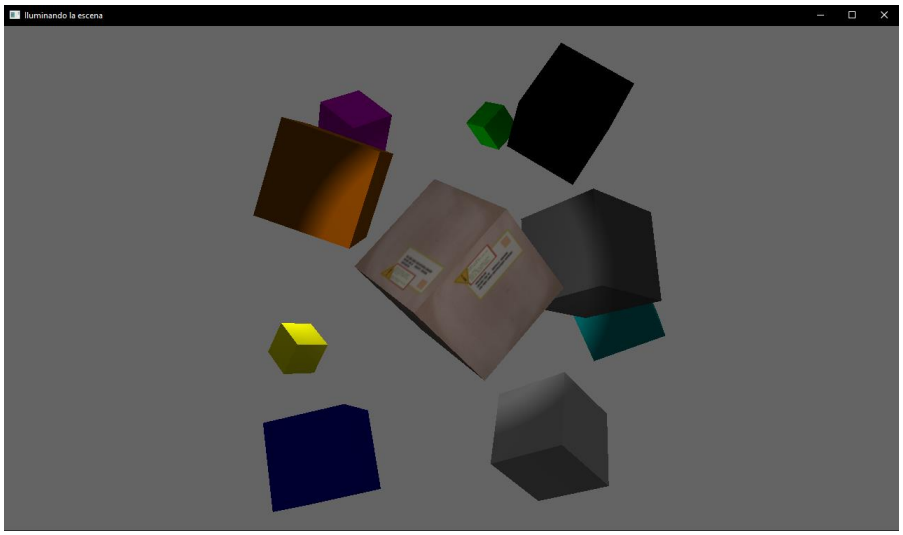
Vale, si te fijas en nuestro processCube cargamos la entidad con el nombre Cubo, claro gracias a nuestro poderosísimo gestor de recursos en verdad este recurso sólo se carga la primera vez, el resto de veces sólo se asigna en memoria.

El color lo gestionamos desde la entidad, por lo que, aunque el recurso sea el mismo el color puede ser diferente. Sin embargo, la textura la gestionamos desde el recurso por lo que todos tendrán la misma textura si la cambiamos. Por eso, vamos a tocar processCube, le vamos a añadir un parámetro para saber si el cubo viene con textura o no y si viene con textura entonces le ponemos un nombre único, diferenciando así los diferentes recursos. Luego actualizamos el main para que sólo el rojo mande el parámetro a true.

[24] Textura Caja

```
// Añadimos un booleano para la textura
Node* createCube(glm::vec3 pos, glm::vec3 size, Color color,
  float rotAngle, glm::vec3 rotAxis, bool texture) {
    auto nodeCube = std::make_unique<Node>();
    nodes.push_back(std::move(nodeCube));
    auto& nCube = nodes[nodes.size() - 1];
    // Si viene con textura, cargamos con un nombre único, para
    // eso uso el tamaño actual del array de modelos, pero tú
    // puedes usar otro sistema si quieres
    if (texture) {
        // Creamos la entidad con un nombre único para evitar
        // compartir texturas entre cubos
        auto entityModel = std::make_unique<E_Model>(
          &RM, "Cubo_" + models.size(), RType::RCube);
        // Lo metemos en el vector de modelos
        models.push_back(std::move(entityModel));
    } else {
        // Creamos la entidad con un nombre compartido para
        // evitar recargar el modelo
        auto entityModel = std::make_unique<E_Model>(
           &RM, "Cubo", RType::RCube);
        // Lo metemos en el vector de modelos
        models.push_back(std::move(entityModel));
    }
    ...

    ...
    // El primer cubo con un true, el resto con false
    Node* cubo1  = createCube(glm::vec3(0.0f, 0.0f, 0.0f),
      glm::vec3(1.0f, 1.0f, 1.0f), Color(1.0f, 0.0f, 0.0f, 1.0f),
      45.0f, glm::vec3(1.0f, 0.0f, 0.0f), true);
    Node* cubo2  = createCube(glm::vec3(2.0f, 5.0f, -15.0f),
      glm::vec3(1.0f, 1.0f, 1.0f), Color(0.0f, 1.0f, 0.0f, 1.0f),
      45.0f, glm::vec3(1.0f, 0.0f, 0.0f), false);
    (etc)
```

Con esto, tu cubo ya tendrá su textura especial mientras que todos los demás tendrán sus colores base.



Ya tienes tu escena iluminada y texturizada. ¡Vaya capítulo! Es muy probable que algo te haya salido mal en algún momento o que lo estés viendo todo, literalmente, muy negro. Para que puedas asegurarte de que todo está bien, como siempre, te dejo el código completo de este capítulo[25].

Si en otros capítulos es útil que juegues con la escena y experimentes, en este es crucial. No sólo por los valores, por cómo se complementan cada parte de tu motor para dibujar e iluminar una escena. Ilumina y retexturiza tu escena que te lo has ganao

[25] Código fuente Fase4

6. COMPLICANDO LAS MALLAS

Enhorabuena, ahora ya tienes realmente el esqueleto de tu motor gráfico. Tienes los cimientos de una casa que podrá ser tan alta como tu quieras, y en este capítulo te lo voy a demostrar. ¿Qué quieres, añadir nuevos tipos, técnicas, shaders...? Tan sólo necesitas aprender sobre el tema que quieras y adaptar tu motor.

Para poner a prueba la potencia de nuestra arquitectura vamos a cargar verdaderos modelos 3D (que ya me estaba cansando de los cubos), vamos a dibujar un cielo y añadir un par de funciones y técnicas que están bien tener. La escalabilidad es clave para cualquier programa, para un motor lo es todavía más. Vamos al lío.

6.1. MODELOS CON ASSIMP

Si STB era nuestro compañero de trabajo a la hora de tratar imágenes 2D, Assimp será nuestro colega para los modelos 3D. Assimp soporta más de 40 formatos de modelos, es una perfecta navaja suiza a la hora de extraer vértices, normales y coordenadas de texturas de un archivo. Sin embargo, aquí el chaval va a necesitar que le echemos un buen cable.

6.1.1. Assimp en Windows

Para poder compilarla en nuestro proyecto me temo que no sólo nos vale con la librería dinámica (dll), también necesitamos la librería estática (lib), vamos a tener que compilarnos nosotros la librería usando CMAKE[26] y el source code de Assimp[27]. Una vez lo hayamos descargado y tengamos CMAKE instalado, abrimos una terminal dentro de la carpeta del source code y vamos a seguir los pasos de Build.md, que es básicamente ejecutar los siguientes comandos:

1. “cd assimp”
2. “cmake CMakeLists.txt”
3. “cmake --build .”

Con esto se nos generarán las carpetas necesarias para nuestro proyecto. Dentro de Include/Assimp estarán las cabeceras que podrás meter dentro de tu carpeta de Librerías, dentro de la carpeta “lib” estará la librería en un archivo .lib, te recomiendo también meterla dentro de Librerías. Por último, dentro de

[26] Descargar CMAKE
[27] Código fuente ASSIMP

“bin” estará la dinámica en un archivo .dll, esta necesitarás ponerla donde se va a generar el ejecutable, en la misma carpeta que tu main.

Ahora que lo tienes todo descargado y en su sitio tendrás que avisarle a tu IDE de que están ahí. Si usas Visual Studio es muy parecido a como instalamos GLFW en el 2.2.1, modificando las propiedades para que en Vinculador/General se referencie al .lib y en Vinculador/Entrada esté el nombre exacto del .dll. Con esto, tu proyecto debería poder incluir Assimp sin problema.

6.1.2. Assimp en Linux

El camino por Linux es muy similar al de Windows, tendrás que compilar con CMAKE[26] el source code de Assimp[27]. La única diferencia es que en vez de generar un .dll generarás un .so y en vez de un .lib un .a, pero los tipos son equivalentes. Recuerda actualizar el makefile para incluir las nuevas librerías, si no al final en el source code del capítulo te dejaré un makefile actualizado.

6.1.3. Leyendo la escena con Assimp

El que va a hacer el trabajo duro va a ser nuestro recurso modelo. Como he dicho antes, Assimp lee la información del archivo y amablemente nos la proporciona bien ordenadita. Sin embargo, es nuestro trabajo procesar toda esa información. Antes de nada, creemos nuestro nuevo recurso en AllResources.

```
// Añadimos uno para un modelo 3D
enum RType{
    RModel, RShader, RCube
};
```

Ahora, en nuestro método load del recurso modelo vamos a contemplar que nos estén creando un recurso con nuestro nuevo tipo.

```
// Copia del vector de texturas cargadas del RM
std::vector<Texture>* texturesLoaded{};
// Directorio de los archivos a cargar
std::string directory{};

// Método de cargado, con la ruta del archivo
// y el tipo de archivo
void load(const std::string& path, std::vector<Texture>&
texturesLoaded, RType rt) override{
    // Asignamos el tipo de recurso
    type = rt;
    // Según el tipo de recursos hace una cosa u otra
    if (rt == RType::RCube) {
        meshes.push_back(processCube());
    }else if (rt == RType::RModel) {
       ...
```

```
        ...
        // Actualizamos el texturesLoaded con el
        // que recibimos del gestor de recursos
        this->texturesLoaded = &texturesLoaded;
        // Iniciamos el importador de Assimp
        Assimp::Importer importer{};
        // Con el lector de Assimp, leemos el modelo y
        // recuperamos la escena, usando las flags que queramos
        const aiScene* scene = importer.ReadFile(path,
             aiProcess_JoinIdenticalVertices |
             aiProcess_Triangulate | aiProcess_GenSmoothNormals|
             aiProcess_FlipUVs);
        // Si no ha habido escena, esta vacía o Assimp nos
        // marca un error, lo mostramos por consola
        if (!scene || scene->mFlags &
          AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode) {
            std::cout << "ERROR::ASSIMP::" <<
                  importer.GetErrorString() << std::endl;
            return;
        }
        // Guardamos la ruta donde está guardado el modelo
        directory = path.substr(0, path.find_last_of("\\/"));
        // Procesamos la información de Assimp
        processNode(scene->mRootNode, scene);
    }
};
```

Como puedes ver, este código se limita a leer la información y devolvérnosla con ese objeto aiScene. Las flags que le pasamos a Assimp no son más que órdenes opcionales que puede seguir. Hay muchas flags[28] pero te voy a explicar las que he usado aquí:

- aiProcess_JoinIdenticalVertices: Identifica y unifica vértices que sean idénticos dentro del modelo, en casos concretos puede ser un problema, pero en la mayoría de casos es mejor tenerlo.
- aiProcess_Triangulate: Convierte cualquier polígono de más de tres lados en triángulos, como hacíamos nosotros con las caras de los cubos.
- aiProcess_GenSmoothNormals: Genera las normales automáticamente, como comprenderás bastante útil.
- aiProcess_FlipUVs: Generalmente las coordenadas de las texturas suelen venir con el eje Y negativo, esto les da la vuelta.

Ahora tendremos que implementar nuestro método processNode. Assimp nos devolverá el modelo 3D como una especie de árbol de escena, donde cada una de las mallas será un nodo que se relacionarán con la jerarquía de un árbol. Por lo tanto, processNode no hará más que ir recorriendo este árbol y procesando las mallas una a una.

[28] Cabecera que contiene la información de todas las flags de Assimp

```
// Por cada una de las mallas del nodo del modelo actualiza el
// vector de mallas y recursivamente hace lo mismo con los hijos
void processNode(aiNode* node, const aiScene* scene) {
    for (unsigned int i{}; i < node->mNumMeshes; i++) {
        // Usamos solo un puntero a la malla para procesarla
        aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
        // Procesamos la malla y la metemos en el vector
        meshes.push_back(processMesh(mesh, scene));
    }
    // Lo mismo para cada hijo del nodo, recorriendo el árbol
    for (unsigned int i{}; i < node->mNumChildren; i++) {
        processNode(node->mChildren[i], scene);
    }
}
```

Ahora viene lo realmente complicado, ese processMesh que vamos a implementar es donde realmente ocurre la magia. El concepto es similar a processCube: Guardar los vértices (con su posición, normales y coordenadas de la textura), índices y texturas en vectores y pasárselos a nuestro recurso malla. Assimp ya nos ha puesto en bandeja esta información sólo tenemos que procesarla ordenadamente.

```
// Procesa la información de la malla de Assimp y
// devuelve un tipo malla propio
R_Mesh processMesh(aiMesh* mesh, const aiScene* scene) {
    // Nuestros vectores para almacenar la info de la malla
    std::vector<Vertex> vertices{};
    std::vector<unsigned int> indices{};
    std::vector<Texture> textures{};
    // Por cada vertice de la malla, procesa su información
    for (unsigned int i{}; i < mesh->mNumVertices; i++) {
        // Creamos el vertex
        Vertex vertex{};
        // Le anadimos la posicion del vertice
        vertex.position.x = mesh->mVertices[i].x;
        vertex.position.y = mesh->mVertices[i].y;
        vertex.position.z = mesh->mVertices[i].z;
        // SI tiene normales, las asigna
        if (mesh->HasNormals()) {
            vertex.normal.x = mesh->mNormals[i].x;
            vertex.normal.y = mesh->mNormals[i].y;
            vertex.normal.z = mesh->mNormals[i].z;
        }
        // Si tiene coordenadas de textura, los asigna
        if (mesh->mTextureCoords[0]) {
            vertex.texCoord.x = mesh->mTextureCoords[0][i].x;
            vertex.texCoord.y = mesh->mTextureCoords[0][i].y;
        }
        // Ahora anade al vector de vertices el vertex leído
        vertices.push_back(vertex);
    }
    ...
```

```
    ...
    // Por cada cara aprovechamos y le sacamos los índices
    for (unsigned int i{}; i < mesh->mNumFaces; i++) {
        // Recuperamos cada cara
        aiFace face{ mesh->mFaces[i] };
        // Assimp ya guarda la información de los índices dentro
        // de la cara, así que la recuperamos y vamos informando
        // el vector de índices
        for (unsigned int j{}; j < face.mNumIndices; j++)
            indices.push_back(face.mIndices[j]);
    }
    // Recuperamos el material desde la escena con el índice
    // que guarda la malla
    aiMaterial* material = scene->mMaterials[
      mesh->mMaterialIndex];
    // Hacemos un vector temporal con todos los mapas de difusión
    std::vector<Texture> diffuseMaps{
      loadMaterialTextures(material, aiTextureType_DIFFUSE,
            "textureDiffuse") };
    // Insertamos al final de textures los mapas difusos, vamos
    // que pegamos el vector de texturas que ha cargado Assimp
    // al final de nuestro propio vector de texturas
    textures.insert(textures.end(),
      diffuseMaps.begin(), diffuseMaps.end());
    // Hacemos lo mismo con las texturas especulares
    std::vector<Texture> specularMaps{ loadMaterialTextures(
      material, aiTextureType_SPECULAR, "textureSpecular") };
    textures.insert(textures.end(),
       specularMaps.begin(), specularMaps.end());
    // Devolvemos un mesh propio con todo
    return R_Mesh(vertices, indices, textures);
}
```

Como ves ya te la he vuelto a liar con las texturas. Sí, vamos a tener que implementar otro método más, el de loadMaterialTextures. No te preocupes, será como estos últimos: Recuperar información de Assimp y cargarla, solo que necesita un pequeño sistema para diferencias nuestros tipos de textura. Le pasamos por parámetro el material leído de Assimp, el enumerado de Assimp para ese tipo de textura y cómo llamamos a nosotros a las texturas en los shaders.

```
// Carga las texturas de los materiales
std::vector<Texture> loadMaterialTextures(aiMaterial* mat,
  aiTextureType type, std::string typeName) {
    // Hacemos un vector local de texturas para luego devolverlas
    std::vector<Texture> textures{};
    // Recuperamos el número de texturas que hay de ese tipo
    if (mat->GetTextureCount(type) != 0) {
        // Como hay textura, el valor de mexcla sera 0
        mixValue = 0.0f;
        // Por cada textura, comprobamos si existe en las
        // texturas cargadas, si no la creamos
        for(unsigned int i{}; i < mat->GetTextureCount(type); i++){
            // Inicializamos la cadena para recuperar el nombre
            aiString str{};
            // Recuperamos la textura
            mat->GetTexture(type, i, &str);
            ...
```

```
            ...
            // Booleano para saber si la ha encontrado
            bool encontrada{};
            // Recorremos nuestro vector de texturas cargadas
            // que antes recuperamos del gestor de recursos
            for(unsigned int j{}; j < texturesLoaded->size(); j++){
                // Comprobamos que los nombres sean iguales
                if (std::strcmp(texturesLoaded->at(j).path.data(),
                  str.C_Str()) == 0) {
                    // Si existe alguna igual, la cargamos y
                    // marcamos como encontrada
                    textures.push_back(texturesLoaded->at(j));
                    encontrada = true;
                    break;
                }
            }
            // Si no lo ha encontrado, la crea
            if (!encontrada) {
                Texture texture{};
                // Cargamos la textura con las rutas de la
                // textura y el modelo
                texture.id = textureFromFile(
                  str.C_Str(), this->directory);
                // Guardamos el tipo, la ruta y lo metemos en
                // nuestro vector local de texturas
                texture.type = typeName;
                texture.path = str.C_Str();
                textures.push_back(texture);
                // Actualizamos el vector de texturas cargadas
                texturesLoaded->push_back(texture);
            }
        }
    }
    return textures;
}
```

Efectivamente, te la he vuelto a liar otra vez, hay un cuarto método llamado textureFromFile a definir. Encapsular todo en métodos pequeños y específicos no sólo es una buena práctica, sino que evitará tener un método monstruosamente largo, así que no nos quejemos tampoco. Vamos a cargar estas texturas en OpenGL como aprendimos en el 5.7. usando la información de Assimp.

```
// Método que recupera la textura según un archivo y un directorio
unsigned int textureFromFile(const char* path,
  const std::string& directory, bool gamma = false) {
    // Ajustamos el path. Assimp lo necesita así
    std::string filename{ std::string(path) };
    filename = directory + '/' + filename;
    // Generamos la nueva textura
    unsigned int textureID{};
    glGenTextures(1, &textureID);
    // Cargamos con stbi la informacion de la imagen
    int width{}, height{}, nrComponents{};
    unsigned char* data = stbi_load(filename.c_str(),
      &width, &height, &nrComponents, 0);
    ...
```

```
    ...
    // Si ha recuperado datos, lo cargamos en la textura si no
    // mostramos un error por terminal
    if (data) {
        // Creamos un formato base, segun el numero de canales
        // de color le ponemos uno u otro
        GLenum format{};
        if (nrComponents == 1)
            // Si solo tiene un canal, solo "rojo"
            format = GL_RED;
        else if (nrComponents == 3)
            // Si tiene tres, RGB
            format = GL_RGB;
        else if (nrComponents == 4)
            // Si tiene cuatro, RGB y un canal alpha
            format = GL_RGBA;
        // Anidamos la textura
        glBindTexture(GL_TEXTURE_2D, textureID);
        // Cargamos la imagen
        glTexImage2D(GL_TEXTURE_2D, 0, format, width,
            height, 0, format, GL_UNSIGNED_BYTE, data);
        // Ponemos los parámetros para los mimaps y las texturas
        glTexParameteri(GL_TEXTURE_2D,
            GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D,
            GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D,
            GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,
            GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        // Generamos los mipmaps
        glGenerateMipmap(GL_TEXTURE_2D);

        // Liberamos la información de los datos
        stbi_image_free(data);
    } else {
        // Debugeamos y nos pasamos el path y el directorio
        // que ha intentado cargar (lo agradecerás)
        std::cout << "Texture failed to load in: " << path
            << " (A.K.A '" << directory << "')" << std::endl;
        stbi_image_free(data);
    }
    // Devolvemos la ID que le ha asignado OpenGL a la textura
    return textureID;
}
```

Realmente como ves no hay nada nuevo, y lo nuevo no es realmente complejo, es sólo que asusta un poco verlo todo junto. Pero como hemos ido construyendo desde abajo, tenemos las bases suficientes para que esto en realidad no parezca tan complicado. Ah, tonto de mí, me he centrado tanto leer los recursos y se me ha olvidado decirte dónde importar Assimp. Lo hacemos en nuestro ResourceManager.hpp.

```
#include <Assimp/Importer.hpp>
#include <Assimp/scene.h>
#include <Assimp/postprocess.h>
```

6.2. MODELOS EN NUESTRA ESCENA

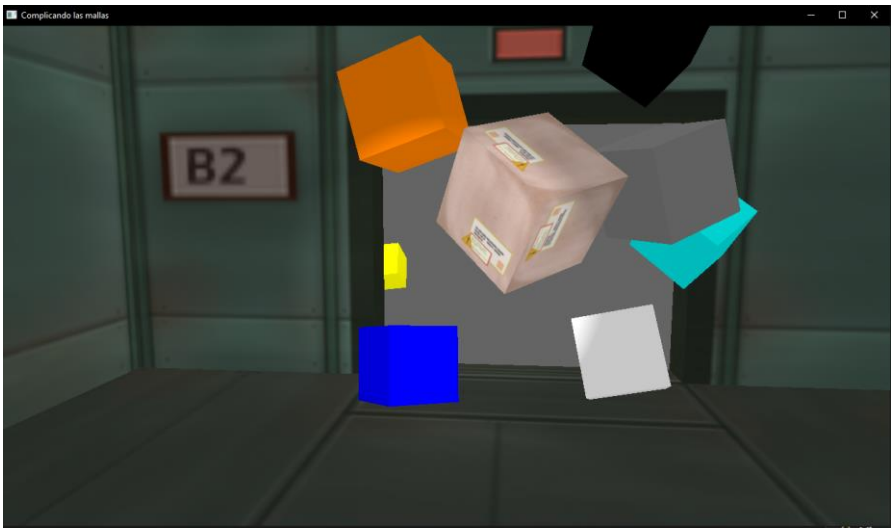
Ahora que ya tenemos nuestros modelos cargados vamos a hacer que existan en nuestra escena. Vamos a tener que hacernos una nueva entidad que... un segundo, en realidad no lo necesitamos. Cuando creamos E_Model ya lo hicimos con todo lo necesario, ya sea para un cubo como para un modelo complejo. Sin embargo, voy a aprovechar para explicarte cómo, en tu Fase5.cpp, vas a crear el nodo en el árbol. Como te imaginarás, el proceso es muy similar al createCube.

```
// Carga del modelo 3D
Node* createModel(std::string path, glm::vec3 pos, glm::vec3 size,
  float rotAngle, glm::vec3 rotAxis, std::string shader) {
    auto nodeModel = std::make_unique<Node>();
    nodes.push_back(std::move(nodeModel));
    auto& nModel = nodes[nodes.size() - 1];
    // Le pasamos nuestro nuevo tipo RModel
    auto entityModel = std::make_unique<E_Model>(
      &RM, path, RType::RModel);
    models.push_back(std::move(entityModel));
    auto& eModel = models[models.size() - 1];
    eModel->setShader("base");
    // Translada el modelo a la posicion por parametro
    nModel->translate(pos);
    // Luego lo rota
    nModel->rotate(glm::vec4(
      rotAxis.x, rotAxis.y, rotAxis.z, rotAngle));
    // Y luego lo escala
    nModel->scale(size);
    nModel->setEntity(std::move(eModel.get()));
    // Le ajusta la relacion de aspecto
    eModel->setAspect(aspect);
    sceneRoot->addChild(std::move(nModel.get()));
    return sceneRoot->getChildren()[
      sceneRoot->getChildren().size() - 1];
}
```

Nada nuevo bajo el sol. Si todo ha ido bien, ya podrías cargar tu primer modelo. Te dejo un modelo de un nivel de prueba[29] y su textura[30]. El modelo está en .obj, esta extensión es muy visual y si te animas a echarle un ojo verás muy fácilmente cómo en verdad no es tan diferente a nuestro cubo. Ahora sólo necesitamos en nuestro main crear los modelos.

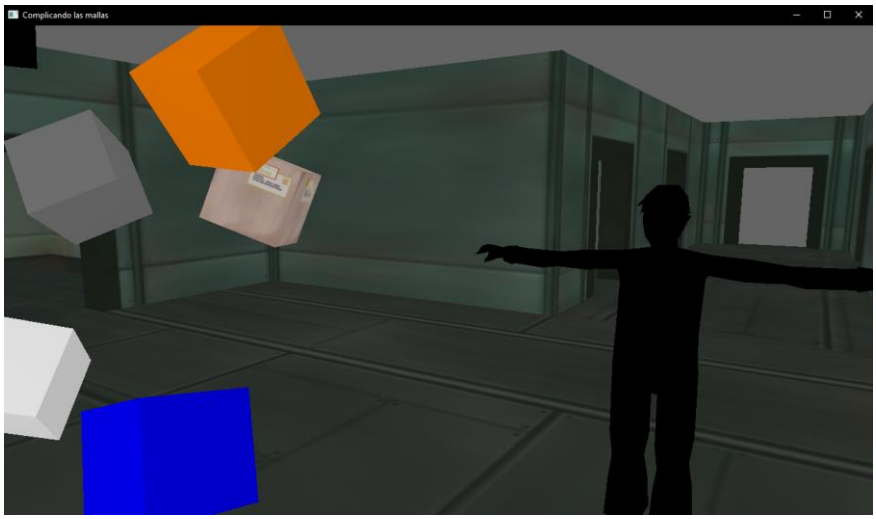
```
// Creamos el modelo del nivel y lo colocamos en el mundo
Node* nivel = createModel("./resources/Mapa.obj",
      glm::vec3(-1.0f, -3.5f, 15.0f),
      glm::vec3(1.0f, 1.0f, 1.0f), 0.0f,
      glm::vec3(0.0f, 1.0f, 0.0f), "base");

// Cargamos la textura para el nivel
dynamic_cast<E_Model*>(
      nivel->getEntity())->setTexture("./Resources/Mapa.png");
```

[29] Modelo del nivel en .obj
[30] Textura del nivel

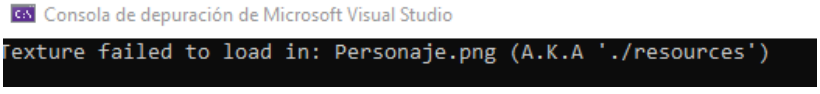


¡Tu primer modelo 3D complejo! Esto es más que una simple caja. ¿Has visto lo fácil que ha sido realmente implementar una funcionalidad completamente nueva en tu motor ahora que ya tienes las bases? Quédate con esa sensación porque es la esencia de este capítulo.

Otros modelos son algo más complejos o vienen encriptados, como el FBX de Autodesk. Esto modelos suelen ser algo más complejos. Mira, te comparto uno[31], intenta cargarlo. Verás una cosa muy graciosa:



Está completamente oscuro, da igual cuánto lo ilumines. Esto es porque el modelo en verdad tenía una textura ya configurada dentro, con una ruta predefinida. Al intentar leerla, nuestro motor nos devolverá un error:



[31] Modelo personaje en .FBX

Con este error podremos darnos cuenta que lo que tenemos que hacer es bajarnos la textura del modelo[32] y ponerla en la ruta que está esperando. En este caso, es en la misma carpeta resources que es donde está el modelo. Volvemos a cargar el modelo, sin necesidad de asignarle nosotros a mano la textura, y verás qué magia.

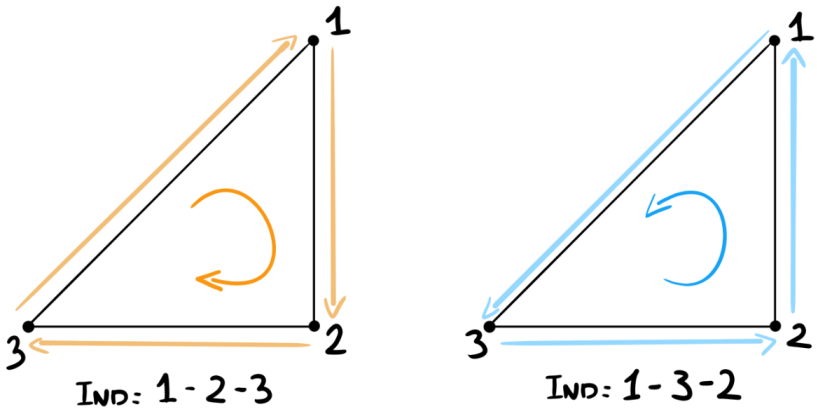
¡Ha cargado! Eso es gracias a nuestro flujo de carga de texturas en nuestro recurso modelo. Hemos usado toda la estructura y ya tenemos un personaje y un nivel, ya no es tanto una escena cubista jeje.

6.3. FACE CULLING

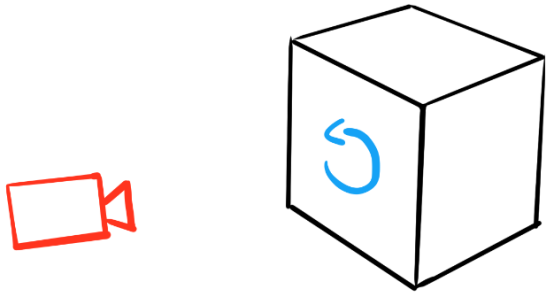
El punto clave es que ahora que tienes los cimientos puedes construir a demanda. ¿Quieres optimizar tu motor? Pues te informas de técnicas que te puedan ayudar y las implementas. Para ejemplificarlo, vamos a implementar una técnica muy sencilla llamada face culling.

El face culling no es más que una técnica de renderizado que se asegura de pintar sólo las caras que estés mirando. OpenGL la tiene desactivada esta función por defecto y es que esta función necesita que tengas un par de cosas en cuenta para que funcione correctamente. La técnica se basa en leer los índices en cierto orden para saber si las caras de tu modelo te están mirando a ti (la cámara) o si te están dando la espalda. Visualmente lo vas a ver mejor.

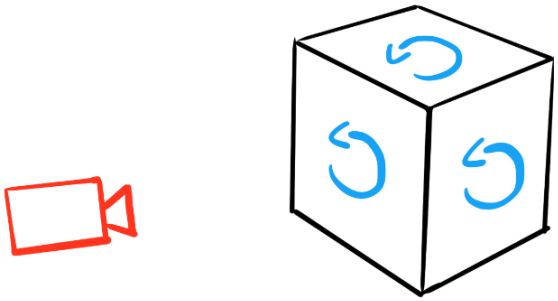
[32] Textura del personaje



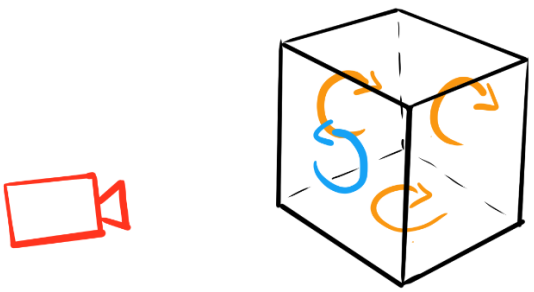
Como ves, el orden de lectura de los vértices puede ser en un sentido horario o antihorario. El face culling de OpenGL, por defecto, esperará que nuestros vértices y nuestros índices estén ordenados en sentido antihorario. ¿En qué ayuda esto a saber si una cara está apuntando a la cámara o no? Muy fácil, con un dibujo lo vas a entender perfectamente.



Como puedes ver la cara que apunta a la cámara gira en sentido antihorario, de hecho, sus hermanas también.



¿Pero y las caras traseras? Pensemos en ellas, si seguimos aplicando el dibujado en sentido antihorario...



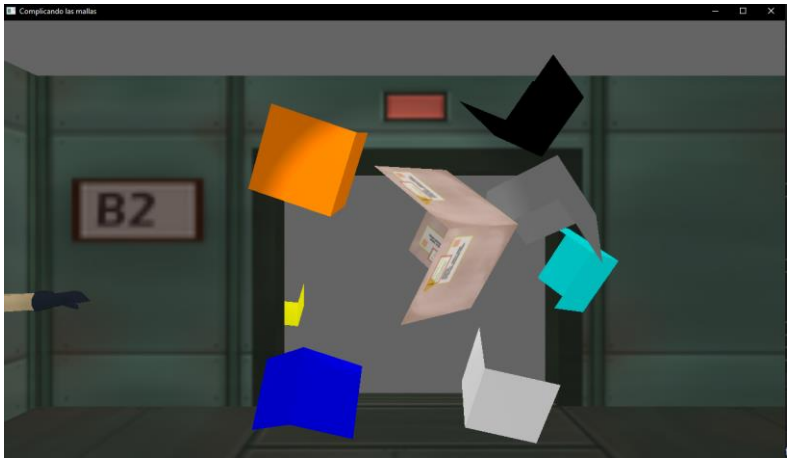
Si tienes un poco de visión te habrás dado cuenta: Nosotros no hemos cambiado el sentido del giro, pero al estar las caras invertidas el sentido de dibujado a pesar de que lo hemos enviado antihorario se ha convertido en horario. Es, evitando dibujar las caras que se lean en sentido horario, cómo funciona el face culling con OpenGL.

Puede parecer que la optimización es del 50% (ahora sólo dibujas 3 caras a la vez), pero como el sentido del dibujado depende de la vista la optimización es variable. Imagina que vieses un cubo perfectamente de frente, te ahorrarías dibujar las otras 5 caras. ¡Eso es más del 50%!

6.3.1. Cambiando el sentido del cubo

Vamos a activar tal cual en nuestro setupOpenGL de la Fase5.cpp el face culling, a ver qué pasa.

```
// Activamos el Face Culling
glEnable(GL_CULL_FACE);
```



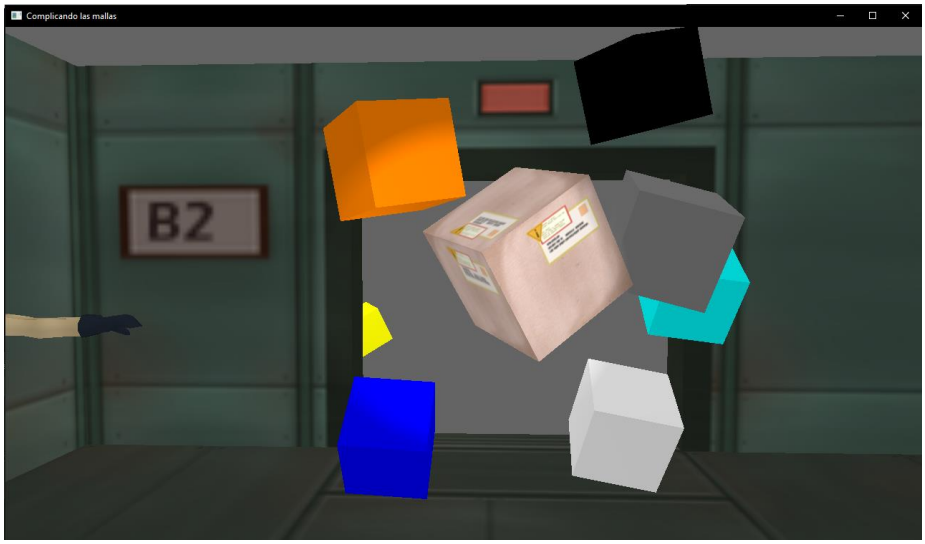
Wow. Claramente algo ha salido muy mal. Bueno, en verdad he sido un poco travieso y cuando creamos el cubo puse mal los índices a propósito. A esto me refería cuando te dije que para el face culling necesitas tener un par de cosas en cuenta.

Assimp ya ha hecho el trabajo de ordenar bien los índices por nosotros, pero para nuestro pobre cubo me temo que tendremos que volver a darle una vuelta. Para hacerlo más claro y visual, he quitado el bucle y los he vuelto a meter directo. Como te dije en su momento, si visualmente no te cuadra, papel y lápiz y te vas dibujando los cubos a mano imitando a OpenGL, ya verás como así aprendes.

```
// Cara superior(y = 0.5),
// en CCW (Contrario a las agujas del reloj)
vertices.push_back(Vertex{ glm::vec3( 0.5f,  0.5f, -0.5f),
      glm::vec3( 0.0f,  1.0f,  0.0f), glm::vec2(1.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3(-0.5f,  0.5f, -0.5f),
      glm::vec3( 0.0f,  1.0f,  0.0f), glm::vec2(0.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3(-0.5f,  0.5f,  0.5f),
      glm::vec3( 0.0f,  1.0f,  0.0f), glm::vec2(0.0f, 0.0f) });
vertices.push_back(Vertex{ glm::vec3( 0.5f,  0.5f,  0.5f),
      glm::vec3( 0.0f,  1.0f,  0.0f), glm::vec2(1.0f, 0.0f) });

// Cara frontal (z = 0.5)
vertices.push_back(Vertex{ glm::vec3( 0.5f, -0.5f,  0.5f),
      glm::vec3( 0.0f,  0.0f,  1.0f), glm::vec2(1.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3( 0.5f,  0.5f,  0.5f),
      glm::vec3( 0.0f,  0.0f,  1.0f), glm::vec2(1.0f, 0.0f) });
vertices.push_back(Vertex{ glm::vec3(-0.5f,  0.5f,  0.5f),
      glm::vec3( 0.0f,  0.0f,  1.0f), glm::vec2(0.0f, 0.0f) });
vertices.push_back(Vertex{ glm::vec3(-0.5f, -0.5f,  0.5f),
      glm::vec3( 0.0f,  0.0f,  1.0f), glm::vec2(0.0f, 1.0f) });

// Cara izquierda (x = -0.5)
vertices.push_back(Vertex{ glm::vec3(-0.5f, -0.5f,  0.5f),
      glm::vec3(-1.0f,  0.0f,  0.0f), glm::vec2(1.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3(-0.5f,  0.5f,  0.5f),
      glm::vec3(-1.0f,  0.0f,  0.0f), glm::vec2(1.0f, 0.0f) });
vertices.push_back(Vertex{ glm::vec3(-0.5f,  0.5f, -0.5f), ç
      glm::vec3(-1.0f,  0.0f,  0.0f), glm::vec2(0.0f, 0.0f) });
vertices.push_back(Vertex{ glm::vec3(-0.5f, -0.5f, -0.5f),
      glm::vec3(-1.0f,  0.0f,  0.0f), glm::vec2(0.0f, 1.0f) });

// Cara inferior(y = -0.5)
vertices.push_back(Vertex{ glm::vec3(-0.5f, -0.5f, -0.5f),
      glm::vec3( 0.0f, -1.0f,  0.0f), glm::vec2(1.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3( 0.5f, -0.5f, -0.5f),
      glm::vec3( 0.0f, -1.0f,  0.0f), glm::vec2(0.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3( 0.5f, -0.5f,  0.5f),
      glm::vec3( 0.0f, -1.0f,  0.0f), glm::vec2(0.0f, 0.0f) });
vertices.push_back(Vertex{ glm::vec3(-0.5f, -0.5f,  0.5f),
      glm::vec3( 0.0f, -1.0f,  0.0f), glm::vec2(1.0f, 0.0f) });

// Cara derecha (x = 0.5)
vertices.push_back(Vertex{ glm::vec3( 0.5f, -0.5f, -0.5f),
      glm::vec3( 1.0f,  0.0f,  0.0f), glm::vec2(1.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3( 0.5f,  0.5f, -0.5f),
      glm::vec3( 1.0f,  0.0f,  0.0f), glm::vec2(1.0f, 0.0f) });
vertices.push_back(Vertex{ glm::vec3( 0.5f,  0.5f,  0.5f),
      glm::vec3( 1.0f,  0.0f,  0.0f), glm::vec2(0.0f, 0.0f) });
vertices.push_back(Vertex{ glm::vec3( 0.5f, -0.5f,  0.5f),
      glm::vec3( 1.0f,  0.0f,  0.0f), glm::vec2(0.0f, 1.0f) });

// Cara trasera (z = -0.5)
vertices.push_back(Vertex{ glm::vec3(-0.5f, -0.5f, -0.5f),
      glm::vec3( 0.0f,  0.0f, -1.0f), glm::vec2(1.0f, 1.0f) });
vertices.push_back(Vertex{ glm::vec3(-0.5f,  0.5f, -0.5f),
      glm::vec3( 0.0f,  0.0f, -1.0f), glm::vec2(1.0f, 0.0f) });
vertices.push_back(Vertex{ glm::vec3( 0.5f,  0.5f, -0.5f),
      glm::vec3( 0.0f,  0.0f, -1.0f), glm::vec2(0.0f, 0.0f) });
vertices.push_back(Vertex{ glm::vec3( 0.5f, -0.5f, -0.5f),
      glm::vec3( 0.0f,  0.0f, -1.0f), glm::vec2(0.0f, 1.0f) });
...
```

```
...
// Vamos cara a cara dibujando los triángulos en orden CCW
indices.push_back(0);indices.push_back(1);indices.push_back(2);
indices.push_back(0);indices.push_back(2);indices.push_back(3);

indices.push_back(4);indices.push_back(5);indices.push_back(6);
indices.push_back(4);indices.push_back(6);indices.push_back(7);

indices.push_back(8);indices.push_back(9);indices.push_back(10);
indices.push_back(8);indices.push_back(10);indices.push_back(11);

indices.push_back(12);indices.push_back(13);indices.push_back(14);
indices.push_back(12);indices.push_back(14);indices.push_back(15);

indices.push_back(16);indices.push_back(17);indices.push_back(18);
indices.push_back(16);indices.push_back(18);indices.push_back(19);

indices.push_back(20);indices.push_back(21);indices.push_back(22);
indices.push_back(20);indices.push_back(22);indices.push_back(23);
```

Con los cubos bien orientados, ahora se dibujan sin problemas. Y con esto ya has aplicado correctamente el face culling, una optimización.

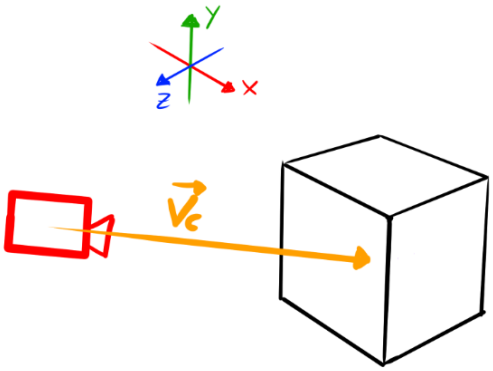


6.4. Siéntete observado

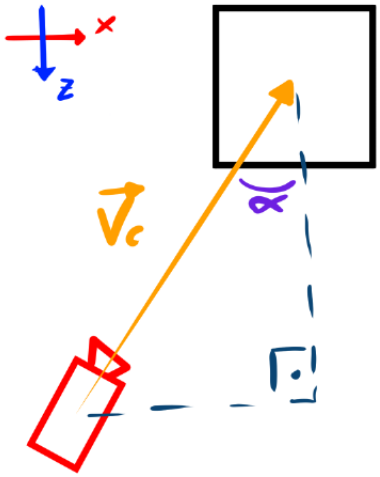
Vale, hemos mejorado un nuevo recurso añadiéndole un tipo nuevo y añadida una optimización, pero, ¿qué tan fácil sería añadirles funcionalidades nuevas a las entidades ya creadas? Se me ocurre, por ejemplo, que el personaje te esté mirando siempre a ti, como si le debieses cinco pavos y te mirase con rencor.

Vamos a implementar un pequeño método llamado lookAt en el que orientaremos un nodo a un objetivo, haciendo que siempre haga la rotación necesaria para encararte da igual dónde estés.

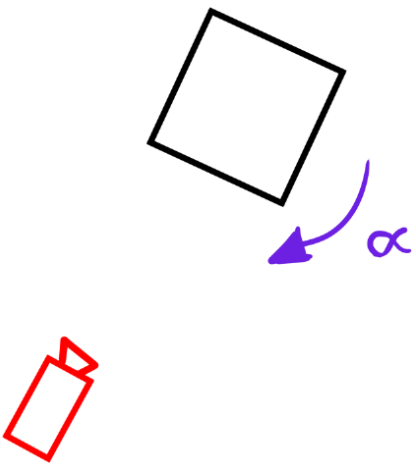
Si sabes de mates (que seguro que sabrás, ¿no?) entonces no será ninguna sorpresa para ti si te digo que para sacar las rotaciones del Yaw y del Pitch que tendrá que hacer el nodo para encararte necesitaremos el poder de las arco tangentes y los arco senos. Lo primero que tenemos que hacer es sacar el vector dirección desde el objeto a la cámara.



Si miramos este vector dirección desde arriba (para ignorar el eje Y) su proyección en el suelo dibujará un triángulo rectángulo en el eje X y Z.

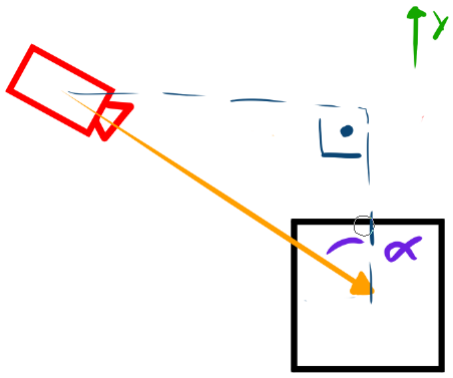


Este ángulo Alpha es el que sacamos con el arco tangente, que además es el ángulo que usaríamos para rotar el Yaw y que el cubo encare a la cámara.

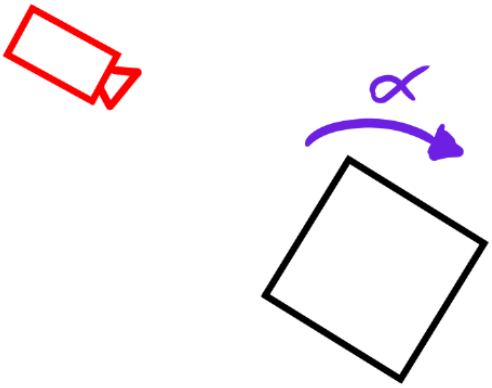


Esto sólo funciona si el cubo de base está apuntando al eje Z positivo, cosa que fácilmente puede no estar ocurriendo. Para solventar esto, tendremos en cuneta un offset para que el usuario pueda ponerle la rotación necesaria para que esté apuntando al eje Z positivo en el caso que no esté inicialmente.

En cuanto al Pitch, es el mismo concepto pero en un plano vertical. Sacamos el ángulo vertical respecto con el vector director y el arco seno



Una vez que tengamos, rotamos el cubo para que encare a la cámara.



En este caso la rotación es sólo de un eje y es absoluta, da igual desde qué rotación parta el objeto calcularemos cuánto necesita rotar para encararlo, no necesitará un offset. Con toda la teoría explicada, pasemos al método, primero lo definiremos en SceneTree.hpp. El método recibirá la rotación base del objeto, ten en cuenta que si el objeto lo ha rotado antes el usuario nosotros tenemos que mantener esa rotación más allá de que lo necesario para que mire a la cámara.

```
void lookAt(glm::vec3 target,
      glm::vec4 baseRotation, float yawOffset = 0.0f);
```

Y lo implementaremos en SceneTree.cpp:

```
// Método que rota el nodo hacia el objetivo, extrayendo
// la posición y escala de la matriz actual
void Node::lookAt(glm::vec3 target, glm::vec4 baseRotation,
float yawOffset) {
    // Extrae la posición de la columna 3 de la matriz
    // de transformación
    glm::vec3 position = glm::vec3(transMatrix[3]);
    // Extrae la escala calculando la longitud de las
    // columnas de la matriz
    glm::vec3 scale = glm::vec3(
        glm::length(glm::vec3(transMatrix[0])),
        glm::length(glm::vec3(transMatrix[1])),
        glm::length(glm::vec3(transMatrix[2]))
    );
    // Calcula la dirección normalizada hacia el objetivo
    glm::vec3 direction = glm::normalize(target - position);
    // Calcula el ángulo yaw (eje Y) y pitch (eje X) a partir
    // de la direccion. Se le pasa el yawOffset por si el
    // modelo base no nace mirando al eje +Z
    float yaw = atan2(direction.x, direction.z) +
      glm::radians(yawOffset);
    float pitch = asin(direction.y);
    // Reconstruye la matriz de transformación desde cero
    transMatrix = glm::mat4(1.0f);
    // Primero recoloca el objeto
    transMatrix = glm::translate(transMatrix, position);
    // Aplicamos la transformación para que acabe mirándote
    transMatrix = glm::rotate(transMatrix, yaw,
      glm::vec3(0.0f, 1.0f, 0.0f));
    transMatrix = glm::rotate(transMatrix, pitch,
      glm::vec3(1.0f, 0.0f, 0.0f));
    // Aplica la orientación inicial del modelo
    transMatrix = glm::rotate(transMatrix,
      glm::radians(baseRotation.w),
        glm::vec3(baseRotation.x, baseRotation.y,
        baseRotation.z));
    transMatrix = glm::scale(transMatrix, scale);
}
```

Lo único que igual te descuadra es que saquemos la posición y la escala de la matriz de transformación. Si quieres recordar por qué, échale un vistazo al 3.3.1.

Si te sientes borracho de poder puedes intentar sacar la rotación base también de la matriz de transformación, pero vas a tener que tener en cuenta que la rotación depende de los ángulos de Euler y es más complicado que simplemente recuperar valores. Además, tendrás que tener en cuenta el Gimbal Lock.

Ahora vamos a actualizar el personaje que cargamos anteriormente para actualizarlo en el bucle de renderizado con nuestra nueva función.

```
personaje->lookAt(sceneRoot->getPrincipalCamera()->
      position, glm::vec4(1.0f, 0.0f, 0.0f, -90.0f), 180.0f);
```

Ahora nuestro personajillo nos sigue con la mirada, qué mono.

6.5. Dibujando el cielo: El Skybox

Igual crees que estoy haciendo trampas simplemente añadiendo mejorando una clase existente, añadiendo una optimización simple y añadiendo una nueva funcionalidad a una entidad existente. Al fin y al cabo, los cambios realmente han sido mínimos. Por eso, voy a subir un poco más las apuestas. Una nueva entidad, nuevos shaders y hasta un array de texturas. Veamos cómo se comporta nuestro motor cuando le añadimos una nueva entidad tan especial.

Lo primero que vamos a necesitar entender es qué es el skybox. Pues es otro cubo, ya sé que a estas alturas estarás cansado de ellos, pero este es especial. Es un cubo que abarca el infinito, piensa en él como el cubo que delimita los bordes de tu pequeño universo 3D. Y al ponerle una textura a este cubo, de repente el universo entero tiene textura, simulando un cielo. Vamos a ir paso a paso.

Crearemos una nueva entidad E_Skybox (recuerda añadirla al AllEntities). Esta entidad fuera del método de dibujado que ya lo explicaré luego no tiene mucho en especial.

```
#pragma once

struct R_Model;
struct R_Shader;
struct E_Camera;

struct E_Skybox : virtual public E_Entity {
      ...
```

```
    ...
    // Constructor del modelo base
    E_Skybox() { skybox = true; };
    // Constructor con un gestor de recurso, necesario para
    // el dibujado, pero sin un modelo
    E_Skybox(ResourceManager* RM_) {
      RM = RM_; skybox = true; };
    // Constructor con un gestor de recurso y un recurso
    // asociado, además de un tipo de recurso
    E_Skybox(ResourceManager* RM_, std::string name,
     RType type) {
      RM = RM_; model = RM->getModel(name, name, type);
      skybox = true; };
    // Metodo de dibujado, de momento no lo vamos a
    // implementar
    void draw(glm::mat4 matrix = glm::mat4(1.0f),
      E_Camera* principalCamera = nullptr) override {
        ...
    };

    // Busca el shader dentro del gesto de recurso y
    // asigna el resultado.
    void setShader(std::string name) {
      shader = RM->getShader(name, name, name); };
    // Setea la relacion de aspecto
    void setAspect(float asp) { aspect = asp; };

    // Un puntero al modelo, el shader y el aspecto ratio
    R_Model* model{};
    R_Shader* shader{};
    float aspect{};
    // El color, por defecto es el amarillo
    glm::vec3 color{ 253.0f, 249.0f, 0.0f };
};
```

Como puedes ver en los constructores, seteamos una nueva variable skybox a true cuando se crea un skybox. Esto es por una razón en específico, los skybox no se dibujan al recorrer el árbol de la escena, como he dicho son especiales, por lo que en la entidad genérica añadimos un booleano llamado skybox que vendrá inicializado a false. De esta forma, todas las entidades lo tendrán, pero sólo cuando se cree un skybox estará a true. Entonces en el método de recorrer el árbol añadimos la comprobación que excluye al skybox.

```
// Eviatmos que se dibuje el skybox
if (entity && entity->RM && !entity->skybox)
    entity->draw(cumulativeMatrix, prCamera);
```

Con esto, ya evitamos que se dibujen al recorrer el árbol. Antes de implementar el método de dibujado de la entidad, vamos a crear dos nuevos Vertex Shader y Fragment Shader para tratar nuestro skybox.

6.5.1. Shaders para el Skybox

Vamos a comenzar con el Vertex Shader. Se parece mucho al primer Vertex Shader que hicimos, con unos pequeños cambios. Nuestro skybox no necesita ni normales ni coordenadas de texturas así que ignoramos los layouts que no usamos.

```
#version 330 core
layout (location = 0) in vec3 aPos;

out vec3 texCoords;

// Matriz de proyección combinada con la vista
uniform mat4 projectionView;

void main(){
    // Las coordenadas de textura del cubo mapa coinciden
    // con la posición del vértice
    texCoords = aPos;
    // Transforma el vértice a la vista de la cámara usando
    // la matriz projectionView
    vec4 pos = projectionView * vec4(aPos, 1.0f);
    // Establece la profundidad a 1.0f (valor máximo) para que
    // el skybox se renderice detrás de todos los objetos
    // Al asignar w a z, tras la división de perspectiva la
    // profundidad será 1.0f
    gl_Position = pos.xyww;
};
```

Lo que hay que entender principalmente es que no necesitamos colocar el cubo en el mundo, por lo que nos olvidamos de la matriz modelo. Las coordenadas irán siempre donde los vértices (no tendría sentido un cielo cuya textura sólo abarcase una parte del horizonte). Por último, al pasar la posición al gl_Position sustituimos el eje Z por el W, lo que al hacer la transformación de proyección siempre dará 1 (W/W = 1) y la profundidad del skybox será la máxima.

Vamos a por el Fragment Shader, es más fácil incluso.

```
#version 330 core
out vec4 FragColor;

in vec3 texCoords;

// No es una simple textura, es una textura cúbica
uniform samplerCube skybox;

void main(){
    // El color es el de la textura, sin más
complicación
    FragColor = texture(skybox, texCoords);
};
```

Ahora que ya tenemos los shaders vamos a meterlos en la carpeta Shaders junto con los básicos y vamos a cargarlos en nuestro main. Voy a aprovechar para cambiar el nombre de los shaders en todo el proyecto, "default" es más correcto e intuitivo que "base", sobre todo ahora que vamos a tener varios shaders. Igual que con nuestro default, guardamos un puntero al skyboxShader.

```
// Nos guardamos un puntero a nuestros shaders
Shader* defaultShader{};
Shader* skyboxShader{};
...

// Leemos los shaders, primero el default
basicShader = &RM.getShader("default",
      "./Shaders/DefaultVertexShader.vs",
      "./Shaders/DefaultFragmentShader.fs")->shader;
// Luego el del skybox
skyboxShader = &RM.getShader("skybox",
      "./Shaders/SkyboxVertexShader.vs",
            "./Shaders/SkyboxFragmentShader.fs")->shader;
```

Ya tenemos los shaders leídos y cargados. Luego sólo tendremos que asignarle a nuestra entidad skybox su shader para que lo use en vez del default. ¿Ves qué fácil es para nuestro motor implementar shaders nuevos que cumplan distintos objetivos? Ahora vamos a por fin dibujar nuestro skybox.

6.5.2. El cielo de la escena

Lo primero que vamos a hacer es guardar el espacio de memoria VRAM para el skybox. No va a ser dinámico porque, hasta donde yo sé, cielo sólo hay uno. Vamos a aprovechar esto para simplemente añadirlo en setupOpenGL, el proceso es muy parecido a cuando lo hacíamos para una textura. Guardamos la ID que nos devuelve OpenGL en una variable global.

```
unsigned int skybox{};
...

glGenTextures(1, &skybox);
// El cambio importante es que usamos el
// enumerado GL_TEXTURE_CUBE_MAP
glBindTexture(GL_TEXTURE_CUBE_MAP, skybox);

// Configuramos el texture sampling
glTexParameteri(GL_TEXTURE_CUBE_MAP,
      GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP,
      GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP,
      GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP,
      GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP,
      GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
```

Ahora que ya tenemos la memoria preparada, vamos a crear la entidad. Hay que tener en cuenta una peculiaridad. No estás cargando una textura, estás cargando seis, una por cara del cubo. Nuestro método createSkybox recibirá un vector con el path de las seis texturas y las cargará una a una.

```
Node* createSkybox(std::vector<std::string> faces) {
    // Enlazamos la textura cúbica que cargamos en el setup
    glBindTexture(GL_TEXTURE_CUBE_MAP, skybox);
    int width{}, height{}, nrChannels{};
    for (uint16_t i{}; i < faces.size(); i++) {
        // Por cada una de sus caras, leemos la textura
        unsigned char* data = stbi_load(faces[i].c_str(),
            &width, &height, &nrChannels, 0);
        // Si la ha leído, añade la textura al cubeMap
        if (data) {
            glTexImage2D(
                // Esto de aquí es importante, índice el orden en el
                // que se van a meter las imágenes en el cubo. El X
                // positivo será el inicio del cubo y el i será el
                // offset que irá aumentando conforme vayamos
                // cargando caras
                GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
                0, GL_RGB, width, height, 0, GL_RGB,
                GL_UNSIGNED_BYTE, data);
        } else {
            // Si no ha cargado debugeamos
            std::cout << "ERROR::SKYBOX::Source skybox
            images cannot load" << std::endl;
        }
        stbi_image_free(data);
    }
    // Ahora que ya hemos cargado las texturas, creamos la entidad
    return loadSkybox();
}
```

El método loadSkybox que tenemos que implementar te sonará mucho a createCube.

```
Node* loadSkybox() {
    auto nodeCube = std::make_unique<Node>();
    nodes.push_back(std::move(nodeCube));
    auto& nCube = nodes[nodes.size() - 1];
    // Es un cubo básico así que lo enviamos con el mismo
    // nombre que el cubo sin textura
    auto entityModel = std::make_unique<E_Skybox>(
      &RM, "Cubo", RType::RCube);
    skyboxModel = std::move(entityModel);
    auto& eCube = skyboxModel;
    // El skybox necesita su shader
    eCube->setShader("skybox");
    eCube->setAspect(aspect);
    nCube->setEntity(std::move(eCube.get()));
    sceneRoot->addChild(std::move(nCube.get()));
    return nCube.get();
}
```

Ahora sí que sí necesitamos crear nuestro método de dibujado de la entidad. No te asustes si ves mucha cosa rara que te lo voy a explicar todo.

```
// Método de dibujado del skybox, renderiza el cubo mapa
// siempre detrás de los objetos de la escena
void draw(glm::mat4 matrix = glm::mat4(1.0f),
  E_Camera* principalCamera = nullptr) override {
    // Si hay un shader, un modelo y una cámara principal
    // se dibuja, si no no se podrá
    if (shader && model && principalCamera) {
        // Asigna el shader al modelo y obtiene la referencia
        model->setShader(&shader->shader);
        Shader& actualShader = shader->shader;
        // Activamos el shader, el del skybox
        actualShader.use();
        // Desactiva el culling de caras para que el
        // skybox se renderice con todas sus caras
        glDisable(GL_CULL_FACE);
        // Desactiva el depth test tradicional para
        // gestionar la profundidad manualmente
        glDisable(GL_DEPTH_TEST);
        // Activa el clamping de profundidad para evitar
        // problemas con la profundidad 1.0f
        glEnable(GL_DEPTH_CLAMP);
        // Usa LEQUAL para que el skybox compita con otros
        // objetos a profundidad máxima
        glDepthFunc(GL_LEQUAL);
        // Extraemos la matriz de vista eliminando la modelo
        // (el skybox sigue a la cámara)
        glm::mat4 view = glm::mat4(
            glm::mat3(principalCamera->getViewMatrix()));
        // Calculamos la matriz de proyección
        glm::mat4 projection = glm::perspective(glm::radians(
            principalCamera->zoom), aspect, 0.1f, 200.0f);
        // Le pasa la combinación de proyección y vista al
        // shader (sin la modelo)
        actualShader.setMat4("projectionView",
            projection * view);
        // Indica al shader que la textura del cubo mapa
        // está en la unidad 0
        actualShader.setInt("skybox", 0);
        // Activa la unidad de textura 0 y vincula el cubo mapa
        // (ID 3 es el skybox precargado, lo pongo a mano por
        // agilidad, pero puedes hacerlo dinámico)
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_CUBE_MAP, 3);
        // Dibuja el modelo usando el shader configurado
        model->drawWithShader(actualShader);
        // Restaura el estado de OpenGL a ç
        // los valores por defecto
        glEnable(GL_CULL_FACE);
        glEnable(GL_DEPTH_TEST);
        glDisable(GL_DEPTH_CLAMP);
        glDepthFunc(GL_LESS);
    }
};
```

Creo que lo único que puede llegar a confundir del método son los cambios de estado de OpenGL que hacemos al principio. Verás, como te dije, estamos

jugando con la profundidad para que el cubo aparezca atrás del todo. Tanto glDisable(GL_CULL_FACE) como glDisable(GL_DEPTH_TEST) quedan lo bastante claro y son caras conocidas. Sin embargo, glEnable(GL_DEPTH_CLAMP) y glDepthFunc(GL_LEQUAL) no sólo son nuevos sino que pueden parecer confusos.

El DEPTH_CLAMP no es más que asegurarnos que esa división de w/w que hace el Vertex Shader dé 1.0 exacto porque, por imprecisión de la coma flotante, podría darse el caso de que fuese mayor. Cualquier profundidad mayor que 1.0 se clipea y no es visible, y nosotros queremos ver nuestro cielo.

Por otro lado, cambiar la función de profundidad a GL_LEQUAL en vez de GL_LESS, como viene por defecto, es necesario porque el skybox nunca se dibujaría de otro modo. Verás, GL_LESS se llama así porque hace que el buffer de profundidad se limpie si es menor que 1.0. Claro, antes hemos clampeado la profundidad a 1.0 por lo que la comprobación será siempre falsa. Lo cambiamos a GL_LEQUAL porque, como su nombre indica, comprobará si es menor o igual a 1.0, solucionando el problema.

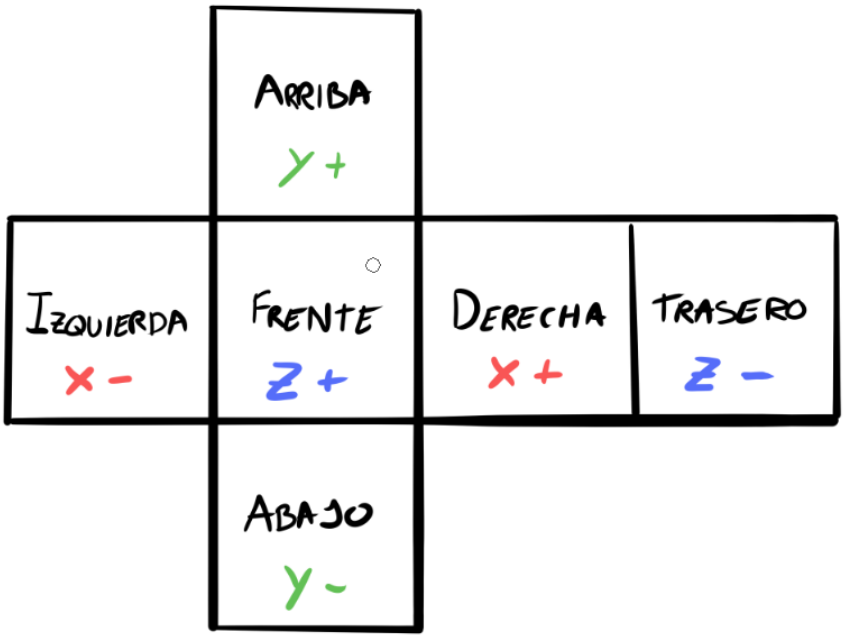
Genial, ya solo nos queda cargar nuestro skybox en el main de Fase5.cpp y renderizarlo en el bucle antes de recorrer el árbol. Te paso yo las texturas del skybox[33], venga.

```
std::vector<std::string> caras{};
caras.push_back("./Resources/Skybox/top.jpg");
caras.push_back("./Resources/Skybox/bottom.jpg");
caras.push_back("./Resources/Skybox/right.jpg");
caras.push_back("./Resources/Skybox/left.jpg");
caras.push_back("./Resources/Skybox/front.jpg");
caras.push_back("./Resources/Skybox/back.jpg");
createSkybox(caras);
...

skyboxModel->draw(glm::mat4{ 1.0f },
      sceneRoot->getPrincipalCamera());
```

Finalmente renderizamos y... bueno, está claro que nuestro cielo está un poco raro.

[33] Texturas del Skybox

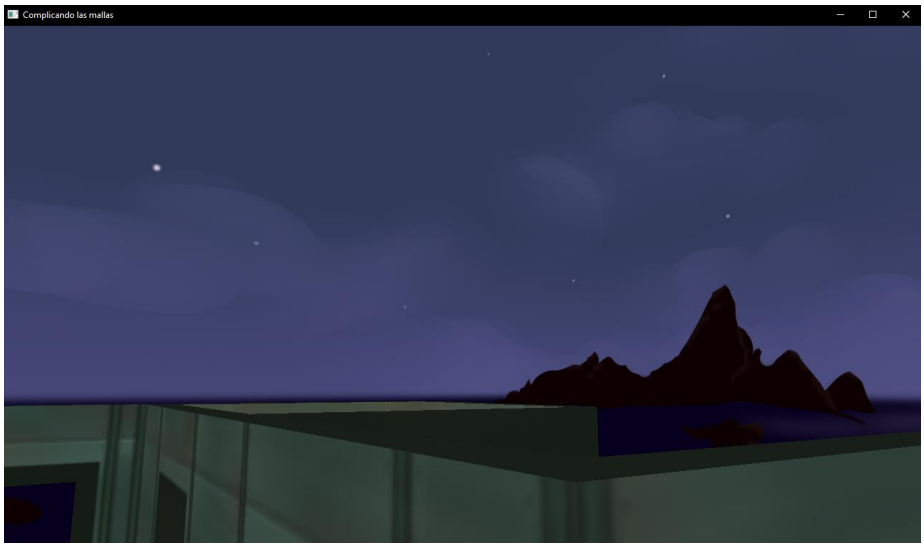
Esto es por el orden en el que hemos cargado las texturas. Como hemos dicho antes, la cara X positiva va a ser la incial. Al ir iterando desde la X positiva (la derecha) pasará a la X negativa (izquierda), después a la Y positiva (arriba), luego la negativa (abajo) y lo mismo con la Z (frente, trasero). Si desplegásemos ese cubo se vería así:



Reordenamos nuestro vector siguiendo esta nueva información:

```
std::vector<std::string> caras{};
caras.push_back("./Resources/Skybox/right.jpg");
caras.push_back("./Resources/Skybox/left.jpg");
caras.push_back("./Resources/Skybox/top.jpg");
caras.push_back("./Resources/Skybox/bottom.jpg");
caras.push_back("./Resources/Skybox/front.jpg");
caras.push_back("./Resources/Skybox/back.jpg");
createSkybox(caras);
```

Y prepárate porque ahora sí que vas a ver el cielo.



Pues ya lo tienes, tu propio cielo cargado con tu propio motor. ¿Has visto como no ha llevado tanto? Esa es la ventaja de unas buenas bases, si quieres expandirte tienes todo el mundo a tu alcance. Sólo necesitas aprender, mejorar y adaptar. Te dejo como siempre el código fuente[34] para que lo revises si quieres.

[34] Código fuente Fase 5

7. FORJANDO EL MOTOR

Has avanzado mucho si has llegado hasta aquí, es un esfuerzo notable. Pero siento decirte que tengo una muy mala noticia. Eso que tienes en tu código no es, para nada, un motor gráfico. No me malinterpretes, no es un mal programa, quizá podría estar mejor optimizado por las libertades pedagógicas que me he tomado pero bueno, al fin y al cabo. Sin embargo, le falta la estructura necesaria para ser llamado motor gráfico con todas las letras.

Vamos deshacer nuestro proyecto y aprovechar sus cenizas para forjar un verdadero motor gráfico y acabar el libro por todo lo alto. Lo primero que debes hacer: Elegir un nombre. Yo, como te imaginarás, lo llamaré Pixeles. Cada vez que me refiera a Pixeles como clase, tú sustitúyelo por tu nombre. Quiero que sea tuyo que para algo lo has construido. Lo segundo que vamos a hacer es crear nuestra propia librería. ¿Recuerdas lo que hicimos con Assimp y CMAKE? Nos bajamos el source code de Assimp y compilamos nuestra librería, ahora vamos a hacer lo mismo, pero para nuestro código.

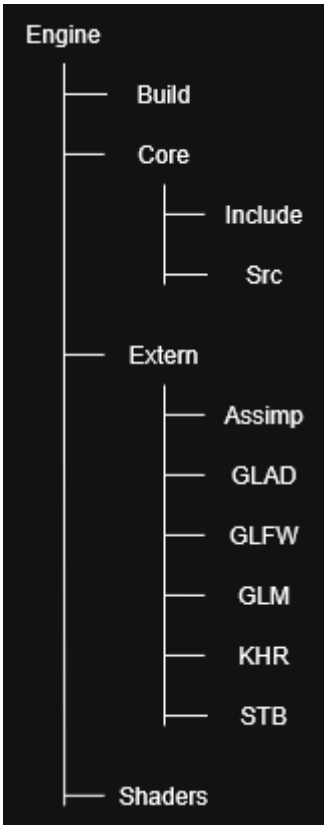
7.1. PUNTOS CLAVE DE UNA LIBRERÍA

Si vamos a rehacer nuestro código necesitamos tener en mente ciertos puntos clave para que no volvamos a caer en hacer un simple programa. Lo primero de todo será adoptar mentalidad pro usuario. Durante el libro te he dado pinceladas al respecto, pero ahora nos lo vamos a tomar en serio. Vamos a ocultar el uso de toda librería que no sea la nuestra:

- ¿Importar GLM, ASSIMP, GLAD y GLFW? Tu usuario va a importar Pixeles y arreglado, todo funcional.
- ¿glm::vec3? Nada de eso, vamos a crear nuestro tipo Vec para que el usuario ni tenga que saber qué es GLM. Y no sólo los vectores, los enumerados de GLFW también.
- ¿glfwWindowShouldClose? Fuera, ocultaremos todo uso de GLFW, OpenGL y demás.
- ¿Devolver Node* al crear entidades? Tu usuario seguramente no habrá visto un árbol en su vida, ni siquiera por la calle, vamos a devolver las entidades como tal
- ¿Guardar las entidades en vectores? Ya lo hacemos nosotros, tu usuario ni se entera que lo estamos haciendo.

¿Ves el patrón verdad? Queremos ponérselo fácil al usuario, que tenga que entender lo mínimo de cómo funciona un motor gráfico realmente para poder usar el tuyo.

Antes de empezar a crear y reorganizar, vamos a hablar de arquitectura. Vamos a usar CMAKE para compilar nuestro proyecto, pero de momento no vamos a hacerle mucho caso, aún así tenemos que tenerlo en cuenta para ver cómo organizamos nuestras carpetas. También tenemos que tener en cuenta que queremos que la configuración de cara al usuario sea la más simple y directa posible. Asumiendo una carpeta Engine (esta la puedes llamar como tu quieras) deberemos tener la siguiente organización:



Con esta configuración, nuestro usuario sólo tendría que incluir la carpeta Inlcude y la de Extern. Vamos a ir una por una, explicando por qué y viendo qué vamos a meter en cada cual:

- Build: Cosas de CMAKE, de momento que se quede vacía
- Core/Include: Tendrá todas nuestras cabeceras, todos los .hpp. Tenerlas todas aquí es lo que permite que luego tu usuario en su configuración del programa, para tener todas las cabeceras del motor, sólo tenga que incluir esta carpeta o mudar el contenido de la misma donde él quiera.

- Core/Src: Los compilados, todos los .cpp. Necesarias para compilar la librería.
- Extern/Assimp: La librería de Assimp, no la modificamos de las fases anteriores. Asegúrate de que tenga las cabeceras y la librería estática.
- Extern/GLAD: Cambia un poco con respecto a cómo lo teníamos en las fases anteriores. Antes esta carpeta tenía otras dos dentro, una include y otra llamada src, si recuerdas de cuando nos la bajamos en el 2.3. Ahora tenemos que mover el contenido de include/glad a la carpeta raíz de GLAD y el de include/KHR a la de Extern. Si no lo hiciésemos así, el usuario debería de incluir además de la carpeta Extern la de GLAD/include.
- Extern/GLFW: Movemos lo del contenido de la carpeta include a la raíz de GLFW. Por el mismo motivo que con GLAD.
- Extern/GLM: No hace falta que la toquemos, solo asegúrate de que tenga todas las cabeceras.
- Extern/STB: Mete aquí la cabecera de STB, para no dejarla suelta en la carpeta.
- Shaders: Simplemente guardará los archivos de los shaders. Aunque te haré un spoiler, no usaremos los de aquí.

Tomate el tiempo de reorganizar las carpetas, reorganiza los includes para que sigan tu nueva estructura. Que todos tus includes tengan <> y no "". No te preocupes si de momento no "las encuentra", luego cuando compile con CMAKE los encontrará. Eso sí, ten en cuenta que todas las cabeceras tienen que ser independientes, si tu cabecera E_Skybox usa glm::vector, tendrá que incluir a GLM, esto con todas y cada una de tus cabeceras. No te preocupes, si se te olvida algo luego cuando compiles te darás cuenta jeje.

En un uso estrictamente correcto, en verdad no deberías exponer todas tus cabeceras sólo la cabecera principal, tus entidades y tus tipos. El resource manager, por ejemplo, podría ir en una carpeta Private dentro de Core. Sin embargo, para un motor pequeño como este y con un objetivo didáctico no está mal.

7.2. ESCONDIENDO LA COMPLEJIDAD

Vamos a crear tres archivos nuevos: Pixel.hpp, Pixel.cpp y PixelTypes.hpp. La cabecera definirá nuestro motor y todas sus capacidades, el compilable sus definiciones y el types guardará todos los tipos especiales de nuestro motor (los que enmascararán las librerías). Vamos a empezar por este último, que será sencillo de entender. De todas formas, al final del apartado te dejaré las clases.

```
#pragma once
// Como usamos GLM, lo incluimos
#include <GLM/glm.hpp>

// Forward declaration de tus entidades
struct Node;
struct E_Camera;
struct E_Entity;
struct E_Model;
struct E_Light;
struct E_Skybox;

// Los tipos de recursos
enum class RType {
    RModel,
    RShader,
    RCube
};

// Nuestro propio tipo color
typedef struct Color {
    float r, g, b, a;

    // Constructores
    Color() : r(0), g(0), b(0), a(0) {}
    Color(float r_, float g_, float b_, float a_) :
       r(r_), g(g_), b(b_), a(a_) {}

    bool operator==(const Color& c) const {
        return (
            r == c.r &&
            g == c.g &&
            b == c.b &&
            a == c.a
            );
    }

    bool operator!=(const Color& c) const {
        return !(*this == c);
    }
} Color;
```

Hasta aquí nada nuevo, vamos con las cosas nuevas.

```
// Colores por defecto, en RGB, define todos los que quieras
#define LIGHTGRAY  Color(200, 200, 200, 1.0f)
#define GRAY       Color(130, 130, 130, 1.0f)
#define DARKGRAY   Color( 80,  80,  80, 1.0f)
...

// Los IDs de las teclas, van en par con las de GLFW,
// yo definiría todas las básicas
#define K_SPACE          32    //   //
#define K_APOSTROPHE     39    // ' //
#define K_COMMA          44    // , //
#define K_MINUS          45    // - //
...
#define K_A              65
#define K_B              66
#define K_C              67
...
```

```
...
// Vectores análogos de glm::vec
typedef struct Vec2 {
    float x, y;

    // Constructores, uno por defecto
    Vec2() : x(0), y(0) {}
    // Otro con parámetros
    Vec2(float x_, float y_) : x(x_), y(y_) {}
    // Y otro que recibe un glm::vec y lo transforma,
    // importantísimo
    Vec2(glm::vec2& v) : x(v.x), y(v.y) {}

    // Todos los operadores
    bool operator==(const Vec2& v) const {
      return (x == v.x && y == v.y); }
    bool operator!=(const Vec2& v) const {
      return (x != v.x || y != v.y); }
    Vec2 operator+(const Vec2& v) const {
      return Vec2(x + v.x, y + v.y); }
    Vec2 operator-(const Vec2& v) const {
      return Vec2(x - v.x, y - v.y); }
    Vec2 operator*(const float& f) const {
      return Vec2(x * f, y * f); }
    Vec2 operator/(const float& f) const {
      return Vec2(x / f, y / f); }
    Vec2 operator+=(const Vec2& v) {
      x += v.x; y += v.y; return *this; }
    Vec2 operator-=(const Vec2& v) {
      x -= v.x; y -= v.y; return *this; }
    // Definimos el operador asignación para glm::vec,
    // también importante
    Vec2& operator=(const glm::vec2& v) {
      x = v.x; y = v.y; return *this; }
} Vec2;

// Lo mismo pero con Vec3 y Vec4
typedef struct Vec3 {
...

typedef struct Vec4 {
...
```

Con esto, ya tenemos nuestros tipos específicos. Como ves con estos ya no tenemos que depender ni de GLFW ni de GLM. Ahora necesitaremos nuestro objeto Pixeles, que va a ser la instancia de nuestro motor. Podrías hacerlo con funciones sueltas, pero a mí me gusta encapsular todo en un objeto, evita complicaciones.

Este objeto Pixeles estará en Pixeles.hpp, tendrá todas las funciones vitales para nuestro motor, algunas utilidades, setups, vectores de entidades... será el corazón de nuestro motor y el que manejará todo en las sombras. De momento declaramos aquí las funciones, ya en Pixeles.cpp las implementaremos.

```
constexpr unsigned int ANCHO{ 1280 };
constexpr unsigned int ALTO{ 720 };
constexpr const char* NOMBRE{ "Pixeles" };

struct Pixeles {
    // Constructor por defecto, por si el usuario quiere
    // instanciar pero no empezar a correr el motor.
    Pixeles() {};
    // Constructo de arranque, con este ya te abre una
    // ventana del motor grafico e inicia el árbol.
    Pixeles(uint16_t width, uint16_t height,
      std::string name);
    // El destructor por defecto y los operadores de copia y
    // asignación destruidos, importante para evitar
    // posibles fugas de memoria.
    ~Pixeles();
    Pixeles(const Pixeles&) = delete;
    Pixeles(Pixeles&&) = delete;
    Pixeles& operator=(const Pixeles&) = delete;
    Pixeles& operator=(Pixeles&&) = delete;

    // Si el usuario ha instanciado el motor con el
    // constructor por defecto, esta función arrancara todo
    // cuando el usuario quiera
    void startWindow(uint16_t width, uint16_t height,
      std::string name);
    void closeWindow();
    // Tu motor no tiene que ser celoso, si ya has abierto
    // una ventana de otra manera, puedes iniciar con el
    // handler de la ventana de OpenGL que hayas inciado tú
    void setWindow(
      void* win, uint16_t width, uint16_t height);
```

Antes de continuar vamos a dejar clara una cosa. Vamos a simplificar el bucle de renderizado. Para lograrlo, vamos a hacer una función start que se deba llamar al principio del bucle (comprobará que haya el mínimo viable para dibujar) y otra llamada end al final (dibujará, cambiará los buffers, etc). De esta forma, tenemos encapsuladas todas las responsabilidades del motor en cuanto a la renderización dejando al usuario sólo la responsabilidad de manejar las entidades en medio si quiere.

```
    // Inicio del renderizado, Asegura unas calidades
    // mínimas para que el renderizado más básico sea visible
    void start();
    // Una vez has creado todo lo que querías renderizar,
    // recorre el árbol y dibuja todo..
    void end();
    // Por si no quiere cambiar los buffers front y back,
    // que no sé pa qué pero mejor dar opciones
    void endNoSwap();
    // Renderiza el árbol, no es mas que una llamada a start
    // y end, pero así el usuario puede controlar el
    // renderizado o solo llamar a renderizar y olvidarse
    void render() { start(); end(); };
    // Limpia el árbol de la escena entero, permite el
    // tratamiento del árbol en un bucle de renderizado.
    void clearScene();
    ...
```

```
        ...
        // Para controlar el bucle de renderizado sin necesidad
        // de llamar directamente a GLFW
        bool rendering() {
              return !glfwWindowShouldClose(mainWindow); };
        void stopRendering(bool cierre) {
              glfwSetWindowShouldClose(mainWindow, cierre); };

        // La función de limpiar el fondo
        void cleanBackground(Color c);
```

Ahora vamos con la creación de entidades. Ten en cuenta lo que dijimos al principio, nada de devolver nodos del árbol, devolvemos las entidades. Luego ahondaremos en la implementación cómo logramos esto, aunque es muy simple. También en la creación de las entidades vamos a meter algunos valores por defecto.

```
        Shader* createShader(std::string name,
              std::string vs, std::string fs);

        // Métodos de creación de entidades renderizadles,
        // con valores por defecto para las versiones "default"
        E_Camera* createCamera(
              Vec3 position = Vec3(0.0f, 0.0f, 0.0f),
              Vec3 up = Vec3(0.0f, 1.0f, 0.0f),
              float yaw = -90.0f,
              float pitch = 0.0f
        );

        // Lo mismo por cada entidad
        E_Light* createDirectionalLight(
              ...
```

Ahora continuamos con los gestores de las propiedades privadas, utilidades... como digo, todo lo necesario.

```
    // Manejamos texturas, tanto si nos pasa nodo como modelo
    void setTexture(E_Model* image, std::string path);
    void setTexture(Node* image, std::string path);

    // Getters para los valores privados
    GLFWwindow* getMainWindow() { return mainWindow; };
    Shader* getDefaultShader() { return defaultShader; };
    Shader* getSkyboxShader() { return skyboxShader; };
    E_Camera* getPrincipalCamera() { return principalCamera; };
    // Si quiere acceder al árbol se lo permitimos,
    // pero es opcional completamente
    Node* getSceneRoot() { return sceneRoot.get(); };
    uint16_t getScreenWidth() { return SCREEN_WIDTH; };
    uint16_t getScreenHeight() { return SCREEN_WIDTH; };

    // Funcionalidades varias, cuando mejores tu motor
    // podrás ir añadiendolas
    bool isKeyPressed(uint16_t key);
    float getTime(){return static_cast<float>(glfwGetTime());};

private:
```

Ya lo último es lo privado, lo que no debería tocar tu usuario de manera directa y lo que usaremos para gestión interna del motor. No hay nada que no hayas visto antes.

```
    // Función interna para el dibujado 3D, básicamente
    // recorre el árbol e informa los uniforms
    void draw3D();

    // Carga el espacio en OpenGL del Skybox
    void setupSkybox();

    // Carga los shaders default
    void loadDefaultShaders();

    // Arranca la ventana de OpenGL y todo lo necesario
    // para nuestro motor
    GLFWwindow* setupOpenGL(uint16_t width,
      uint16_t height, std::string name);

    // Carga las imágenes de un skybox con OpenGL
    void loadSkybox(std::vector<std::string> faces);

    // Métodos para pasar de nuestras clases propias a GLM
    glm::vec2 parseGLM(Vec2 vector) {
      return glm::vec2(vector.x, vector.y); }
    glm::vec3 parseGLM(Vec3 vector) {
      return glm::vec3(vector.x, vector.y, vector.z); }
    glm::vec4 parseGLM(Vec4 vector) {
      return glm::vec4(vector.x, vector.y, vector.z,vector.w);}

    // El puntero único de la raíz del nodo
    std::unique_ptr<Node> sceneRoot{};
    // Punteros a los shaders básicos,
    // a la camara principal y a la ventana principal
    Shader* defaultShader{};
    Shader* skyboxShader{};
    E_Camera* principalCamera{};
    GLFWwindow* mainWindow{};
    // Instancia del gestor de recursos
    ResourceManager RM{};
    // Vectores para almacenar los nodos, necesario para
    // gestionar y encapsular las responsabilidades de la
    // destrucción de la memoria
    std::vector<std::unique_ptr<Node>> nodes{};
    std::vector<std::unique_ptr<E_Camera>> cameras{};
    std::vector<std::unique_ptr<E_Light>> lights{};
    std::vector<std::unique_ptr<E_Model>> models{};
    std::unique_ptr<E_Skybox> skyboxModel{};
    // Numero de luces
    // X -> Luces direccionales
    // Y -> Luces puntuales
    // Z -> Luces especulares
    glm::vec3 numLights{};
    // Booleano para ver si está activo el motor o no
    bool active{}, skyLoaded{};
    uint16_t SCREEN_WIDTH{}, SCREEN_HEIGHT{};
    unsigned int skybox{};
};
```

Y ya está, tu clase tiene declarado todo lo necesario para funcionar como hasta ahora, puede que incluso mejor. Vamos con la implementación. Como mucho de esto ya lo hemos visto en realidad, voy a ir al grano y sólo hablaré de las cosas nuevas o antiguas que necesitan cambios.

```
// Variables generales para la ventana, la relación de aspecto
// y el ultimo color de limpieza de pantalla
GLFWwindow* window{};
float aspect{};
Color cleanColor{};
unsigned int WIDTH{}, HEIGHT{};

// Importante para manejar nosotros la memoria y no haya fugas
Pixeles::~Pixeles() = default;

// Constructor que arranca Pixeles
Pixeles::Pixeles(uint16_t width,
  uint16_t height, std::string name) {
    // Inicializamos la pantalla principal con OpenGL
    mainWindow = setupOpenGL(width, height, name);
    // Guardamos nuestras variables locales y globales del motor
    WIDTH = width;
    HEIGHT = height;
    SCREEN_WIDTH = width;
    SCREEN_HEIGHT = height;
    // Calculamos el aspect ratio y lo guardamos
    aspect = static_cast<float>(width) /
      static_cast<float>(height);

    // Si esta pantalla no existe es que OpenGL ha fallado por
    // algún motivo, abortamos y pintamos por pantalla
    if (mainWindow == NULL) {
        std::cout << "ERROR::OPENGL::FAILED TO CREATE WINDOW"
            << std::endl;
        return;
    }
    // Le asignamos a la global window la ventana
    window = mainWindow;
    // Creamos el nodo raíz del árbol
    sceneRoot = std::make_unique<Node>();
    // Le decimos al resource manager que esta es la raíz de
    // nuestra escena
    RM.setScene(sceneRoot.get());
    // Precargamos el shader base
    loadDefaultShaders();
    // Creamos el espacio para el skybox en OpenGL
    setupSkybox();
}

void Pixeles::loadDefaultShaders(){
    defaultShader = &RM.getShader("default",defaultVertexShader,
      defaultFragmentShader, false)->shader;
    skyboxShader = &RM.getShader("skybox", skyboxVertexShader,
      skyboxFragmentShader, false)->shader;
}
```

Si eres muy perspicaz habrás visto que he añadido un booleano a la creación del shader. Sí, por esta razón es por la que separé el método de getResource genérico e hice uno especial para los shaders.

Vamos a cambiar el flujo, ahora el constructor de nuestra clase Shader puede aceptar un booleano que le indicará si carga el shader desde un archivo (como hasta ahora) o si lo hará desde un string. ¿Por qué? Bueno, si dependiésemos de que el usuario tenga los shaders en una carpeta específica es muy probable que algo inesperado pase y no se carguen bien los shaders. Lo mejor es que los incrustemos como una cadena en un ShadersDefault.hpp para luego importarlo en nuestro Pixeles.cpp.

```
#pragma once

const char* defaultVertexShader = R"(
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
...
};
)";

const char* defaultFragmentShader = R"(
#version 330 core

out vec4 FragColor;
...
};
)";

// Lo mismo con los del skybox
```

De esta forma conseguimos dos cosas. La primera es que nuestros shaders siempre carguen independientemente de la ruta relativa de los archivos y la segunda es que si el usuario quiere cargar sus propios shaders como un string ahora nuestro motor se lo permite pasándole false al constructor. Vamos a añadirle la declaración nueva a Shader.hpp y la implementación a nuestro Shader.cpp.

```
// Shader.hpp
// Constructor con el path del VertexShader y el FragmentShader
Shader(const char* vertexPath, const char* fragmentPath, bool
isFile = true);

void loadShaderFromFile(const char* vertexPath, const char*
fragmentPath);
void loadShaderFromString(const char* vShaderCode, const char*
fShaderCode);
```

```
// Shader.cpp
Shader::Shader(const char* vertexPath, const char* fragmentPath,
bool isFile){
    if(isFile){
        loadShaderFromFile(vertexPath, fragmentPath);
    }else{
        loadShaderFromString(vertexPath, fragmentPath);
    }
}

void Shader::loadShaderFromFile(const char* vertexPath, const
char* fragmentPath){
    // Leemos el archivo, cargamos los datos...
    // lo que ya teníamos
}

void Shader::loadShaderFromString(const char* vShaderCode, const
char* fShaderCode){
    // No hace falta que carguemos los datos,
    // directamente se los pasamos a OpenGL con el Stirng
}
```

Ahora volvamos a nuestro Pixeles.cpp y a nuestros métodos de renderizado.

```
// Inicio del dibujado, comprueba que haya una cámara al menos
void Pixeles::start() {
    if (!principalCamera)
        createCamera();
}

// Final del dibujado, se encarga de renderizar
void Pixeles::end() {
    // Mínimo que haya una luz direccional
    if (lights.size() == 0)
        createDirectionalLight();
    // Llamamos al método de dibujado global
    draw3D();
    glfwSwapBuffers(mainWindow);
    glfwPollEvents();
}

void Pixeles::draw3D() {
    for (auto& l : lights)
        l->draw(glm::mat4{ 1.0f },
        sceneRoot->getPrincipalCamera());

    defaultShader->setInt("n_directionalLights",
      static_cast<int>(numLights.x));
    defaultShader->setInt("n_pointLights",
      static_cast<int>(numLights.y));
    defaultShader->setInt("n_spotLights",
      static_cast<int>(numLights.z));
    // Sólo pintamos el skybox si hay
    if (skyboxModel)
      skyboxModel->draw(glm::mat4{ 1.0f },
      sceneRoot->getPrincipalCamera());

    // Recorremos el árbol
    sceneRoot->traversal(glm::mat4{ 1.0f },
      sceneRoot->getPrincipalCamera());
}
```

La creación de entidades va a cambiar un poco. Lo primero que tenemos que tener en cuenta es que el usuario base si no quiere no tiene por qué trabajar con nodos. Vamos a tener que guardar un puntero al nodo en las entidades y hacer métodos que invoquen a los de transformación del nodo, pero usando nuestros vectores y no los de GLM. Para que los tengan todos, lo haremos E_Entity.hpp.

```
// E_Entity.hpp
Node* node{};
void translate(Vec3 toTranslate);
void rotate(Vec4 toRotate);
void scale(Vec3 toScale);
void lookAt(Vec3 target, Vec4 baseRotation, float yawOffset);

// SceneTree.cpp
void E_Entity::translate(Vec3 toTranslate) {
    node->translate(glm::vec3(toTranslate.x, toTranslate.y,
toTranslate.z));
}
void E_Entity::rotate(Vec4 toRotate) {
    node->rotate(glm::vec4(toRotate.x, toRotate.y, toRotate.z,
toRotate.w));
}
void E_Entity::scale(Vec3 toScale) {
    node->scale(glm::vec3(toScale.x, toScale.y, toScale.z));
}
void E_Entity::lookAt(Vec3 target, Vec4 baseRotation, float
yawOffset) {
    node->lookAt(
        glm::vec3(target.x, target.y, target.z),
        glm::vec4(baseRotation.x, baseRotation.y,
            baseRotation.z, baseRotation.w),
        yawOffset
    );
}
```

Ahora tenemos que tener cuidado de que todos los métodos de creación añadan el nodo a la entidad.

```
E_Camera* Pixeles::createCamera(Vec3 position, Vec3 up, float
  yaw, float pitch) {
    // Cargamos la entidad como siempre
    ...
    // Recuerda actualizar los vectores internos para la memoria
    cameras.push_back(std::move(entityCam));
    ...
    if(sceneRoot->getPrincipalCamera() == NULL){
        sceneRoot->setPrincipalCamera(eCam.get());
        // Recuerda para la camara y el Skybox actualizar
        // las variables internas del motor
        principalCamera = eCam.get();
    }
    ...
    // Guardamos el nodo en la entidad
    eCam->node = nCam.get();
    ...
    // Devuelve un puntero casteado a la entidad especÃfica
    return dynamic_cast<E_Camera*>(eCam.get());
}
```

Todos los demás métodos son prácticamente idénticos, solo que traducidos a su nuevo entorno. Con esta nueva estructura si quisieras añadir una nueva función, como por ejemplo poner la ventana en pantalla completa, lo primero sería declarar el método dentro de nuestro objeto Pixeles y luego añadir su implementación.

```
// Pixeles.hpp
void switchFullscreen();

// Pixeles.cpp
// Para saber si está o no en pantalla completa
bool fullscreen{};

// Va cambiando entre pantalla completa o modo ventana
void Pixeles::switchFullscreen() {
    if (fullscreen)
        glfwSetWindowMonitor(window, NULL, 0, 0,
            WIDTH, HEIGHT, 60);
    else
        glfwSetWindowMonitor(window, glfwGetPrimaryMonitor(), 0,
            0, WIDTH, HEIGHT, 60);
    fullscreen = !fullscreen;
}
```

Ya tienes tu motor con la estructura de un motor hecho y derecho. Ampliable, modulable y que esconde su complejidad al usuario para que el mismo necesite saber lo mínimo posible. Te dejo los tres archivos[35] por si hay algo que no te ha quedado del todo claro. Ahora con todo cambiado vamos a generar, por fin, nuestra librería con CMAKE.

7.3. COMPILANDO CON CMAKE

Para que CMAKE compile todo por nosotros, necesita que le digamos cómo trabajar. Es un poco vago. Para esto necesita que le hagamos unos TXT llamados CMakeLists.txt. Concretamente, haremos dos: Una en la carpeta raíz de Engine que configurará el proyecto en general y otra en la de Core que se encargará de que nuestros includes y el source code se compile y se linkee bien con las librerías.

El CMakeLists de Engine será muy sencillo, simplemente le dirá que el importante es el de Core.

[35] Pixeles.hpp, Pixeles.cpp y PixelesTypes.hpp

```
cmake_minimum_required(VERSION 3.15)
project(PixelesWorkspace)

# Le decimos a CMAKE que mire en Core
add_subdirectory(Core)
```

El de Core es el que da más miedo, pero línea a línea irá teniendo más sentido.

```
cmake_minimum_required(VERSION 3.15)
project(PixelesEngine VERSION 1.0.0 LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# ${CMAKE_CURRENT_SOURCE_DIR} es desde donde está el
# archivo, es decir Engine/Core/

# Guardamos en EXTERNAL_DIR la ruta a las dependencias
# Extern, las librerias
set(EXTERNAL_DIR ${CMAKE_CURRENT_SOURCE_DIR}/../Extern)

# Vamos añadiendo nosotros a mano las cabeceras
# y librerías para enlazarlas
set(GLFW_INCLUDE_DIR ${EXTERNAL_DIR}/glfw/include)
set(GLFW_LIBRARY ${EXTERNAL_DIR}/glfw/lib/glfw3.lib)

set(ASSIMP_INCLUDE_DIR ${EXTERNAL_DIR}/assimp/include)
set(ASSIMP_LIBRARY ${EXTERNAL_DIR}/assimp/lib/assimp.lib)

set(GLM_INCLUDE_DIR ${EXTERNAL_DIR}/glm/include)
set(GLAD_INCLUDE_DIR ${EXTERNAL_DIR}/glad/include)

set(STB_INCLUDE_DIR ${EXTERNAL_DIR})

# Le decimos dónde está el source code, que están dentro
# de Src, archivo a archivo
set(ENGINE_SOURCES
    ${CMAKE_CURRENT_SOURCE_DIR}/Src/Pixeles.cpp
    ${CMAKE_CURRENT_SOURCE_DIR}/Src/Shader.cpp
    ${CMAKE_CURRENT_SOURCE_DIR}/Src/SceneTree.cpp
    ${CMAKE_CURRENT_SOURCE_DIR}/Src/ResourceManager.cpp
    ${EXTERNAL_DIR}/glad/src/glad.c
)

# Archivos de cabecera, importante si quieres que
# tu IDE los reconozca
set(ENGINE_HEADERS
    ${CMAKE_CURRENT_SOURCE_DIR}/Include/Pixeles.hpp
    ${CMAKE_CURRENT_SOURCE_DIR}/Include/PixelesTypes.hpp
    ${CMAKE_CURRENT_SOURCE_DIR}/Include/Shader.hpp
    ${CMAKE_CURRENT_SOURCE_DIR}/Include/ShadersDefault.hpp
    ${CMAKE_CURRENT_SOURCE_DIR}/Include/SceneTree.hpp
    ${CMAKE_CURRENT_SOURCE_DIR}/Include/E_AllEntities.hpp
    ${CMAKE_CURRENT_SOURCE_DIR}/Include/E_Entity.hpp
    ${CMAKE_CURRENT_SOURCE_DIR}/Include/E_Camera.hpp
    ${CMAKE_CURRENT_SOURCE_DIR}/Include/E_Light.hpp
    ${CMAKE_CURRENT_SOURCE_DIR}/Include/E_Model.hpp
    ${CMAKE_CURRENT_SOURCE_DIR}/Include/E_Skybox.hpp
    ...
```

```
    ...
    ${CMAKE_CURRENT_SOURCE_DIR}/Include/ResourceManager.hpp
    ${CMAKE_CURRENT_SOURCE_DIR}/Include/R_Resource.hpp
    ${CMAKE_CURRENT_SOURCE_DIR}/Include/R_Shader.hpp
    ${CMAKE_CURRENT_SOURCE_DIR}/Include/R_Model.hpp
    ${CMAKE_CURRENT_SOURCE_DIR}/Include/R_Mesh.hpp
    ${CMAKE_CURRENT_SOURCE_DIR}/Include/R_AllResources.hpp
)

# Creamos la librería estática
add_library(PixelesEngine STATIC ${ENGINE_SOURCES}
${ENGINE_HEADERS})

# Directorios de inclusión, los de nuestra cabecera y las
librerías
target_include_directories(PixelesEngine
    PUBLIC
        ${CMAKE_CURRENT_SOURCE_DIR}/Include
        ${GLM_INCLUDE_DIR}
        ${GLAD_INCLUDE_DIR}
        ${STB_INCLUDE_DIR}
    PRIVATE
        ${GLFW_INCLUDE_DIR}
        ${ASSIMP_INCLUDE_DIR}
)

# Linkamos las librerías para que no las tenga que enlazar el
usuario
target_link_libraries(PixelesEngine
    ${GLFW_LIBRARY}
    ${ASSIMP_LIBRARY}
    opengl32
)
```

Con CMAKE ya configurado, vamos a la carpeta Build tan vacía que teníamos antes. Ejecutamos los siguientes comandos en la terminal:

- “cmake ..”
- “cmake --build . --config Release” o “cmake --build . --config Debug”

Ten cuidado con lo de Release/Debug, luego cuando hagamos el entorno de pruebas tendremos que compilarlo también en Release o Debug según hayamos compilado la librería (con tu usuario también, pero ya es cosa suya). Si todo ha ido bien, aunque es probable que a la primera no te haya salido, dentro de Build/Core/Release (o Debug) esté tu tan amado .lib/.a.

Si no ha funcionado, la consola te habrá dicho por qué. Si no encuentras forma de solucionar el problema o te estás calentando mucho la cabeza, recuerda que al final del capítulo tendrás el source code completo de la Fase 6 donde podrás revisar y contrastar el código que necesites.

Los dos comandos que necesitará el usuario para compilar tu librería, junto con explicaciones varias, los dejaremos en un readme o por el estilo en la carpeta Engine. Es una buena práctica. ¡Ya tenemos nuestra librería, enhorabuena! Recuerda que cuando vayas a distribuir la tuya, dejes limpia la carpeta Build. Hay quien incluso la borra por completo y ya que el usuario se la cree.

7.4. UN USUARIO ENTRA EN LA ESCENA CUBISTA

Vamos a recrear una última vez nuestra escena cubista, pero esta vez vamos a imitar que somos un usuario que se ha bajado un .zip con tu librería y ha metido la carpeta Engine al lado de su main.cpp (en nuestro caso, Fase6.cpp). Lo primero que tendremos que hacer es, si estamos en Windows, configurar el IDE, en nuestro caso Visual Studio. Si estamos en Linux con hacer bien el makefile o incluir las librerías vamos sobrados. (recuerda cómo en el 2.2.1 y 2.2.2)

Lo primero que nos daremos cuenta es de que, gracias a nuestra arquitectura, el usuario sólo deberá incluir en la configuración el "Engine/Core/Include" y el "Engine/Extern". Con eso, el usuario ya tiene todo. Al linkar sí que va a tener que incluir la ruta donde ponga el .lib/.a de tu librería y las rutas "Engine\Extern\GLFW\lib" y "Engine\Extern\Assimp\" ya que ambas tienen el .lib/.a de sus respectivas librerías. Además, en las dependencias adicionales, hasta ahora teníamos glfw3.lib, opengl32.lib y assimp-vc145-mtd.lib, pues ahora tenemos que añadir orgullosamente nuestro PixelesEngine.lib. Un último detalle es que el usuario tendrá que tener los dll de Assimp y de GLFW en la misma ruta donde se va a crear su ejecutable, asegúrate de distribuirlos con tu motor y de añadir en el detalle el readme.

Hay formas de evitar que tenga que linkar estas librerías, pero complican demasiado la explicación para nuestro pequeño primer motor.

Y, finalmente, recreamos nuestra escena cubista.

```
// Sólo hace falta incluir Pixeles
#include <Pixeles.hpp>

// Variables para deltaTime
float lastFrame{};
float deltaTime{};

// Instanciamos el motor
Pixeles MG{ 1280, 720, "Tu Primer Motor Gráfico" };

// Guardamos un puntero a nuestra cámara principal
E_Camera* principalCamera{};
...
```

```
// Procesamos los inputs, pero con nuestro motor
void processInput() {
    // Es lo mismo, pero escondemos GLFW
    if(MG.isKeyPressed(K_W)) {
        principalCamera->processKeyboard(FORWARD, deltaTime);
    }
    ...
}

// Lo del ratón y el zoom
float lastX{}, lastY{};
bool firstMouse{ true };

void movimientoRaton(GLFWwindow* window, double xPosIn, double
yPosIn) {
    // Esto es igual
    ...
    // Usamos el puntero a nuestra cámara principal,
    // aunque podríamos usar el del motor
    if (principalCamera != NULL)
        principalCamera->processMouseMovement(xOffset, yOffset);
}

void zoomRaton(GLFWwindow* window, double xOffset, double
yOffset) {
    // Igual, pero con el matiz de la cámara
}

int main() {
    // Creamos la cámara principal
    principalCamera = MG.createCamera(Vec3(-0.3f, 0.0f, 5.0f),
      Vec3(0.0f, 1.0f, 0.0f), -90.0f, 0.0f);

    // Seteamos lo del ratón (No lo he escondido para ir rápido,
    // pero podrías hacerlo si quieres)
    glfwSetCursorPosCallback(MG.getMainWindow(),
      movimientoRaton);
    glfwSetScrollCallback(MG.getMainWindow(), zoomRaton);
    glfwSetInputMode(MG.getMainWindow(), GLFW_CURSOR,
      GLFW_CURSOR_DISABLED);

    // Creamos las luces con nuestro motor, como ves ya no es
    // necesario que le pasemos todos los parámetros
    MG.createDirectionalLight();
    MG.createPointLight(
        Vec3(-3.8f, 0.0f, -12.3f), Vec3(0.2f, 0.2f, 0.2f),
        Vec3(0.8f, 0.8f, 0.8f), Vec3(1.0f, 1.0f, 1.0f),
        Vec3(1.0f, 0.09f, 0.032f));
    auto* spotLight = MG.createSpotLight(
        principalCamera->getPosition(),
        principalCamera->getFront(),
        Vec3(0.0f, 0.0f, 0.0f), Vec3(1.0f, 1.0f, 1.0f),
        Vec3(1.0f, 1.0f, 1.0f), Vec3(1.0f, 0.09f, 0.032f),
        Vec2(12.5f, 15.0f));

    // Creamos los cubos
    std::vector<E_Model*> cubos{};
    ...
```

```
...
E_Model* cubo1  = MG.createCube(Vec3(0.0f, 0.0f, 0.0f),
  Vec3(1.0f, 1.0f, 1.0f), Color(1.0f, 0.0f, 0.0f, 1.0f),
  45.0f, Vec3(1.0f, 0.0f, 0.0f), true);
...

// Nos guardamos los cubos para iterar luego
cubos.push_back(cubo1);
...

// Cargamos la textura para el cubo rojo, el primero
cubo1->setTexture("./Resources/Caja.png");

// Creamos el modelo del nivel y lo colocamos en el mundo
E_Model* nivel = MG.createModel("./resources/Mapa.obj",
  Vec3(-1.0f, -3.5f, 15.0f), Vec3(1.0f, 1.0f, 1.0f),
  0.0f, Vec3(0.0f, 1.0f, 0.0f));

// Cargamos la textura para el nivel
nivel->setTexture("./Resources/Mapa.png");

// Cargamos el personaje
E_Model* personaje = MG.createModel(
  "./resources/Personaje.fbx", Vec3(-5.0f, -1.5f, 0.0f),
    Vec3(0.01f, 0.01f, 0.01f),-90.0f,Vec3(1.0f, 0.0f,0.0f));

// Cargamos el skybox
std::vector<std::string> caras{};
caras.push_back("./Resources/Skybox/right.jpg");
caras.push_back("./Resources/Skybox/left.jpg");
caras.push_back("./Resources/Skybox/top.jpg");
caras.push_back("./Resources/Skybox/bottom.jpg");
caras.push_back("./Resources/Skybox/front.jpg");
caras.push_back("./Resources/Skybox/back.jpg");
MG.createSkybox(caras);

// Inicializamos el bucle, usando nuestro motor
while (MG.rendering()) {
    // Iniciamos el start de nuestro motor
    MG.start();
    // Controlamos el input del usuario
    processInput();
    // Limpiamos el fondo
    MG.cleanBackground(
        Color(100.0f, 100.0f, 100.0f, 255.0f));

    // Calculamos el deltaTime
    // (Esto podría hacerlo tu motor si quieres)
    float currentFrame = MG.getTime();
    deltaTime = currentFrame - lastFrame;
    lastFrame = currentFrame;

    // Iteramos por cada cubo
    for (unsigned int i = 0; i < 10; i++) {
        // Esto sigue igual
    }
    ...
```

```
        ...
        // Actualizamos la posición y dirección de la luz
        // con la de la cámara
        spotLight->setPosition(principalCamera->getPosition());
        spotLight->setDirection(principalCamera->getFront());

        // Que nuestro personaje nos persiga con la mirada
        personaje->lookAt(
            principalCamera->getPosition(),
            Vec4(1.0f, 0.0f, 0.0f, -90.0f), 180.0f);

        // Finalizamos y renderizamos
        MG.end();
    }

    // Cerramos la ventana cuando termine
    MG.closeWindow();
    return 0;
}
```

Si ejecutas este programa verás lo que vería tu usuario: Pura magia.

¡Ahora sí que tienes un motor gráfico, con todas las de la ley! Distribuible, escalable, simple... Es el esqueleto perfecto para ir construyendo a partir de aquí un motor capaz de rivalizar con cualquiera del mercado. Te dejo el código como siempre porque es probable que lo necesites.[36]

Hay muchas cosas a mejorar, eso sí, como he ido repitiendo me he tomado ciertas libertades didácticas. Pero la capacidad de mejorar este pequeño proyecto está en tus manos. Vamos a cerrar el viaje con un pequeño resumen de lo aprendido.

[36] Código fuente Fase 6

8. DIARIO DE VIAJE

Has llegado al final o, mejor dicho, al principio de tu viaje. Quizá llegaste sin saber qué era OpenGL y has salido tras haber construido, paso a paso, tu propio motor funcional desde cero. Si echas una vista atrás, verás todo lo que has conseguido:

1. Has creado, configurado y controlado tu primera ventana tras configurar GLFW, GLAD y GLM.
2. Has dominado los shaders, los buffers, los vértices y los índices para crear una escena llena de cubos bajo tu control.
3. Rehiciste por primera vez tu trabajo para seguir una verdadera arquitectura de un motor gráfico, con un árbol de la escena y un gestor de recursos. No sólo eso, has poblado tu mundo con entidades nueva como la cámara, que ya se podía mover libre.
4. Has, y ojo que lo digo pronto, iluminado tu mundo con Phong y un Fragment Shader mucho más profundo. Has creado tus tres luces y no contentos con eso les has metido texturas con STB.
5. Como te veías fuerte has cargado modelos complejos con ASSIMP y has puesto a prueba la capacidad de escalabilidad de tu motor, la cual ha superado con creces.
6. Para poner la cereza sobre el pastel, has vuelto a rehacer tu trabajo, pero esta vez aprovechando de verdad la arquitectura de un motor gráfico y has generado tu propia librería.

Tienes delante de ti un esqueleto muy robusto de un motor gráfico, pero lo más importante es que lo has construido tú, entendiendo qué hace cada parte y el porqué de su función.

8.1. MIRANDO HACIA EL FUTURO

Como digo, tienes una base muy buena, te voy a dar un par de ideas para que puedas aprovecharlas y crecer. En el ultimo punto te he guardado una sorpresa.

8.1.1. Las sombras

Créeme, si ya te parecía complicado proyectar luz sobre tu mundo de OpenGL, las sombras están a otro nivel. Sin embargo, es de esas cosas que consiguen sacarte una sonrisa una vez ves a tu primer cubito proyectar una sombrita.

El concepto es simple, renderizar la escena desde el punto de las luces para ver qué puntos chocan y cuáles no... ¿He dicho simple? Me habré equivocado. En fin, te dejo una buena referencia[37] y una presentación que hice para una ponencia hace unos años[38], aunque sin la explicación pierde un poco.

8.1.2. Transparencias

Nuestro tipo color admite un Alpha, pero no le hemos dado mucho uso. Las transparencias necesitan controlar muy bien el orden, al fin y la cabo si renderizas primero un objeto transparente y otro opaco después no va a saber no vas a poder recalcular los pixeles ya pintados del primero. El truco está en separar los transparentes y controlar el orden de dibujado.

8.1.3. Animaciones

Si eres un valiente, ASSIMP permite cargar huesos y animaciones de un modelo que las tenga. Hacer tu propio motor de animaciones puede ser la cosa más divertida o la más frustrante del mundo. Hay dias que ver al muñeco bailar de forma anormal por haber calculado algo mal te hace gracia y otras quieres tirar el portátil por la ventana. Eso sí, cuando lo ves bailar bien, casi te dan ganas de bailar a ti también.

8.1.4. Carga de niveles

Como has visto sería sencillo cargar la información de una escena de un archivo y simplemente meterla entre un start y un end. Esto simularía un cargador de niveles, lo dificil está en decidir cómo tratas la información, pero tú eres el motor, tú pones las reglas, luego que los usuarios se apañen a hacerlo como tú lo has pedido.

[37] Artículo de LearnOpenGL sobre Shadow-Mapping
[38] Presentación Shadow Mapping

8.1.5. Un pequeño motor llamado Inazuma

Resulta que en mis tiempos mozos yo hice mi primer motor gráfico en Linux que tiene algunas de las mejoras mencionadas. Aunque hay algunas cosas que he mejorado para Pixeles, sigue su misma filosofía. Si te interesa echarle un ojo para ver cómo podrías ampliar tu motor, es todo tuyo[39].

8.2. BIBLIOGRAFÍA

Durante el libro nos hemos apoyado en muchos recursos magníficos y libres, que han contribuido a mi desarrollo como programador y, ahora, al tuyo. Te los dejo por aquí.

8.2.1. Bibliografía del libro

1. Learncpp: Tutorial exhaustivo de C++ moderno. https://www.learncpp.com/
2. LearnOpenGL: La referencia por excelencia para aprender OpenGL moderno. https://learnopengl.com/
3. OpenGL Tutorials: Versión en español de los tutoriales de OpenGL. https://www.opengl-tutorial.org/es/
4. GLFW: Página oficial y descargas. https://www.glfw.org/
5. GLAD: Servicio web para generar el cargador de funciones OpenGL. https://glad.dav1d.de/
6. GLM: Repositorio de la librería matemática. https://github.com/g-truc/glm
7. STB: Colección de librerías de dominio público de Sean Barrett. https://github.com/nothings/stb
8. Assimp: Repositorio de la librería de importación de modelos. https://github.com/assimp/assimp
9. CMAKE: Herramienta de construcción multiplataforma. https://cmake.org/
10. Euler (gimbal lock) Explained: Vídeo que visualiza el problema del gimbal lock. https://www.youtube.com/watch?v=zc8b2Jo7mno
11. Camera Rotation With Quaternions: Introducción al uso de cuaterniones para la cámara. https://www.youtube.com/watch?v=MZuYmG1GBFk
12. Guías de Red Hat sobre uso de librerías en GCC. https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/developer_guide/gcc-using-libraries

[39] Motor Gráfico Inazuma

8.2.2. Bibliografía recomendada

13. Real-Time Rendering (Tomas Akenine-Möller, Eric Haines, Naty Hoffman): Libro de referencia sobre técnicas avanzadas de renderizado en tiempo real.
14. Game Engine Architecture (Jason Gregory): Visión completa de la arquitectura de un motor de videojuego.
15. The Book of Shaders: Introducción interactiva a los shaders. https://thebookofshaders.com/
16. RenderDoc: Herramienta de depuración de gráficos imprescindible. https://renderdoc.org/
17. Presentación sobre Shadow Mapping: Mi ponencia donde explico paso a paso cómo añadir sombras a un motor OpenGL. https://github.com/Oscarfcb4/Pixeles/blob/main/ShadowMapping.pptx
18. Motor Inazuma: Mi primer motor gráfico que tiene algunas expansiones comparado con Pixeles como sombras y animaciones. https://github.com/Oscarfcb4/Pixeles/tree/main/Inazuma/Linux

8.3. GLOSARIO

- API (Application Programming Interface): Conjunto de funciones que permiten interactuar con un sistema. OpenGL es una API gráfica.
- Assimp (Open Asset Import Library): Biblioteca para cargar más de 40 formatos de modelos 3D y obtener vértices, normales, texturas, etc.
- Atenuación: Fenómeno por el cual la intensidad de una luz disminuye con la distancia. En luces puntuales y focales se modela con una fórmula que depende de coeficientes constante, lineal y cuadrático.
- Buffer: Bloque de memoria en la GPU donde se almacenan datos (vértices, índices, profundidad, etc.).
- CMAKE: Sistema de construcción multiplataforma que usamos para compilar la librería Pixeles.
- Coordenadas homogéneas: Añadir una cuarta componente (w) a un vector 3D para poder representar traslaciones mediante multiplicación de matrices.
- Cross product (producto vectorial): Operación entre dos vectores que devuelve un tercero perpendicular a ambos. Muy útil para construir el vector right o up de la cámara.
- Delta Time: Tiempo transcurrido entre el frame anterior y el actual. Esencial para que el movimiento sea independiente de la velocidad de los FPS.

- Dot product (producto escalar): Operación entre dos vectores que devuelve un valor relacionado con el coseno del ángulo que forman. Base de los cálculos de iluminación difusa y especular.
- EBO (Element Buffer Object): Buffer que almacena los índices para reutilizar vértices y ahorrar memoria.
- FOV (Field of View): Ángulo de visión de la cámara. Un FOV más grande muestra más escena, pero con más deformación en los bordes.
- Face culling: Técnica que descarta las caras de un modelo que no están orientadas hacia la cámara, ahorrando trabajo a la GPU.
- Fragment Shader: Programa que se ejecuta por cada fragmento (píxel potencial) y determina su color final.
- GLAD: Librería que carga las funciones de OpenGL de forma dinámica, resolviendo las direcciones según el driver instalado.
- GLFW: Biblioteca para crear ventanas, manejar entradas de teclado/ratón y contextos OpenGL.
- GLM (OpenGL Mathematics): Biblioteca de matemáticas diseñada para trabajar con GLSL, con tipos como vec3, mat4 y funciones como lookAt, perspective, etc.
- Índices: Números que indican en qué orden se deben dibujar los vértices para formar triángulos, permitiendo reusar los mismos vértices en múltiples triángulos.
- Interpolación: Proceso por el cual la GPU calcula los valores intermedios de un atributo (color, normal, coordenada de textura) entre los vértices de un triángulo.
- LookAt: Matriz que coloca la cámara en una posición y la orienta hacia un punto de interés.
- Luz ambiental: Componente de iluminación que simula la luz rebotada que llega a todas las superficies sin dirección específica.
- Luz difusa: Componente que depende del ángulo entre la normal de la superficie y la dirección de la luz. Aporta el sombreado básico.
- Luz especular: Componente que produce el brillo concentrado, dependiendo del ángulo entre la luz reflejada y la dirección de la vista.
- Malla (mesh): Conjunto de vértices e índices que definen la geometría de un objeto.
- Mipmap: Colección de versiones preescaladas de una textura que mejoran el rendimiento y la calidad al visualizar objetos lejanos.
- MixValue: En nuestro motor, valor entre 0 y 1 que controla la mezcla entre la textura y el color base del material.
- Modelo de Phong: Modelo de iluminación que combina luz ambiental, difusa y especular para obtener un resultado realista.

- Normal: Vector unitario perpendicular a una superficie en un punto. Fundamental para los cálculos de iluminación.
- OpenGL: Especificación de una API gráfica multiplataforma que permite comunicarse con la GPU.
- Pipeline gráfico: Secuencia de etapas por las que pasan los datos desde que llegan a la GPU hasta que se convierten en píxeles en pantalla.
- Profundidad (Z-Buffer): Buffer que almacena la distancia de cada fragmento a la cámara para decidir si se dibuja o se descarta.
- Rasterización: Etapa del pipeline que convierte los triángulos (vectores) en fragmentos (píxeles).
- Shader: Pequeño programa que se ejecuta en la GPU dentro del pipeline gráfico. Los principales son el Vertex Shader y el Fragment Shader.
- Skybox: Cubo que envuelve la escena y al que se le aplica una textura que simula el cielo o el entorno lejano.
- STB (stb_image): Librería de un solo archivo para cargar imágenes en formatos comunes (JPEG, PNG, etc.).
- Textura: Imagen 2D que se "pega" sobre la superficie de un modelo para darle detalle.
- Uniform: Variable global en un shader que puede ser modificada desde el programa C++ y que mantiene su valor constante para todos los vértices/fragmentos durante una llamada de dibujo.
- VAO (Vertex Array Object): Objeto que almacena la configuración de los VBOs y los punteros de atributos, facilitando el cambio entre diferentes modelos.
- VBO (Vertex Buffer Object): Buffer que almacena los datos de los vértices (posición, normal, coordenadas de textura, etc.).
- Vertex Shader: Primer shader programable del pipeline, ejecutado por cada vértice. Se encarga de transformar las posiciones 3D al espacio de pantalla.
- Viewport: Región de la ventana donde se dibuja la escena. Se configura con glViewport.

Si has llegado hasta aquí, gracias. Me alegra haber podido acompañarte en tu camino. ¡Te toca a ti forjar tu futuro!